BSSC(2000)1 Issue 1

C and C++ Coding Standards

Prepared by:
ESA Board for Software
Standardisation and Control
(BSSC)

european space agency / agence spatiale européenne
8-10, rue Mario-Nikis, 75738 PARIS CEDEX, France

DOCUMENT STATUS SHEET

DOCUMENT STATUS SHEET			
1. DOCUMENT TITLE: BSSC(2000)1			
2. ISSUE	3. REVISION	4. DATE	5. REASON FOR CHANGE
1	0	2000	

Approved, March 30th 2000
Board for Software Standardisation and Control
M. Jones and U. Mortensen, BSSC co-chairmen

TABLE OF CONTENTS

PREFACE

This Coding Standard is based upon the experience of developing custom space system software using the C and C++ programming languages.

The BSSC wishes to thank the Analysis and Verification Section (TOS-MCV) of the European Space Research and Technology Centre (ESTEC), Noordwijk, The Netherlands, and in particular Duncan Gibson, as well as the Software Engineering and Standardisation section (TOS-EME) and in particular Maurizio Martignano, for preparing the standard. The BSSC also thank all those who contributed ideas for this standard. The BSSC members have reviewed standard: Michael Jones (co-chairman), Uffe Mortensen (co-chairman), Alessandro Ciarlo, Daniel de Pablo and Lothar Winzer.

Requests for clarifications, change proposals or any other comments concerning this standard should be addressed to:

BSSC/ESOC Secretariat
Attention of Mr M Jones
ESOC
Robert Bosch Strasse 5
D-64293 Darmstadt
Germany

BSSC/ESTEC Secretariat
Attention of Mr U Mortensen
ESTEC
Postbus 299
NL-2200 AG Noordwijk
The Netherlands

CHAPTER 1
INTRODUCTION

1.1 SCOPE AND APPLICABILITY

International ISO standards already exist for C (ISO/IEC 9899:1990) and C++ (ISO/IEC 14882). These standards define the language constructs of C and C++. Many books and documents describe how these features can be used. These texts usually describe what is possible and not necessarily what is desirable or acceptable, especially for large software engineering projects intended for mission- or safety-critical systems.

This document provides a set of guidelines for programming in C and C++ which are intended to improve the overall quality and maintainability of software developed by, or under contract to, the European Space Agency. The use of this standard should improve consistency across different software systems developed by different programming teams in different companies.

The guidelines in this standard should be met for C or C++ source code to fully comply with this standard. The standard has no contractual implication. Contractual obligations are given in individual project documents.

1.2 DEFINITIONS, ACRONYMS AND ABBREVIATIONS

- A **guideline** is a rule or recommendation, which is applied to the production of code.
- A **rule** is a mandatory practice, which shall be applied during production. Any exceptions shall be documented.
- A **recommendation** is a practice, which should be applied during production but is not mandatory. It is not necessary to document all places where a recommendation is not followed.

CHAPTER 1
INTRODUCTION

This page is intentionally left blank

CHAPTER 2
DOCUMENT OVERVIEW

This document is intended to build on the output of the "Software Top-Level Architectural Design", "Design of Software Items" and "Coding and Testing" phases (ECSS-E-40 terminology) and follows a "top-down" approach so that guidelines can begin to be applied as soon as detailed design of software items starts and before any code is produced. Chapter 4 provides a summary of the general guidelines as stated in the earlier BSSC Software Standard PSS-05-05 "Guide to the software detailed design and production phase". Subsequent chapters describe the specific guidelines to be applied to the production of C and C++ code.

All rules and recommendations will be numbered for reference purposes.

Each rule shall have a short definition followed by a paragraph justifying the application of the rule. The rule may also contain an example of how to apply it, or the consequences of not applying it.

Each recommendation shall have a short definition, which may be followed by a justification and example. There is no need for a list of exceptions because a recommendation is not mandatory.

All rules will be printed in **bold type**.

All recommendations will be printed in *italic type*.

All source code will be printed in a `fixed width font`.

CHAPTER 2
DOCUMENT OVERVIEW

This page is intentionally left blank

CHAPTER 3
THIS DOCUMENT AND THE SOFTWARE ENGINEERING STANDARDS

3.1 ADVICE FROM THE EARLIER PSS-05-05 STANDARD

The BSSC Software Standard PSS-05-05 "Guide to the software detailed design and production phase", in section 2.5, states that understandability of code can be achieved in a variety of ways:

- introducing an introductory header for each module [which contains a title, a configuration item identifier, the original author, the creation date and the change history];
- declaring all variables;
- documenting all variables;
- using meaningful, unambiguous names;
- avoiding mixing data types;
- avoiding temporary variables;
- using parentheses in expressions;
- laying out code legibly;
- adding helpful comments;
- avoiding obscuring the module logic with diagnostic code;
- adhering to structured programming rules;
- being consistent in the use of the programming language;
- keeping modules short;
- keeping the code simple.

The following chapters will provide necessary definitions and more specific guidelines for most of the principles outlined above, although not in the same order. The order of the guidelines is not strictly "top-down" from the overall system down to the smallest detail because some issues are intertwined. For example, comments are used at all levels within the system so they warrant their own section rather than repeating the same information throughout the guidelines.

3.2 POSITION OF THIS DOCUMENT WITH RESPECT TO THE ECSS-E40

The ECSS family of standards is organised around families. In particular, the Engineering family has a dedicated number for software (40). The ECSS-E40 document (Space Engineering - Software) recalls the various software engineering processes and list requirements for these processes in terms of activities that have to be performed and pieces of information that have to be produced. ECSS-E40 does not address directly the coding standards, but requires that the coding standards are defined and agreed, at various levels of the development, between the customer and the supplier.

In particular, the selection of this C/C++ standard could be the answer to the following requirements of the standard ECSS-E-40A (Space Engineering – Software, 13 April 1999):

5.2.2.1 System Requirements, Expected Output e) : Identification of lower level software engineering standards that will be applied [RB;SRR]

5.3.2.11 Each supplier shall define the Software Engineering standards that he intends to follow for his application area (Expected Output b) [RB, SRR]

5.4.2.1 Software requirements, Expected Output i) : identification of lower level software engineering standards that will be used [TS;PDR].

CHAPTER 4
GENERAL PROJECT GUIDELINES

Rule 0. Never break a rule without documenting it.

This rule if obviously a meta-rule, and hence this is the reason why it is numbered as 0.

The complete set of rules is intended to provide consistency across all C and C++ source code but this assumes that there are no constraints upon the programming environment. This is obviously not always the case and compromises may need to be made to handle specific requirements or restrictions of the operating system, or the compiler, or third party code and libraries, etc.

If a rule is broken on a project-wide basis then the reason for breaking the rule should be clearly laid out in the project documentation. However if the rule is only violated in a few specific places within the source code then this should be clearly documented at those places. Anyone who examines the code at a later point will then be able to see why the rule was broken. It is to be hoped that there will also be some review process in which deviations from the guidelines will be examined.

Rule 1. Maintain the source code and associated files under a configuration control system and document CM information in the files.

During the development process it is not uncommon for design changes to be made which mean that newer components are incompatible with older components. The same is true of software development and indeed the rate of modification of software components is usually a lot higher than physical components. Software is also usually developed in an incremental way where each release of the software offers the user additional functionality. After each "major" release it is common for there to be a series of "minor" releases which correct deficiencies in the major release so that the software can still be used until the next major release is available.

Each software release must be assembled using compatible software components otherwise the user is likely to experience problems. It is therefore important that the software developers ensure that they know which components are compatible, and that all versions of components are available for building the different releases of the software. In modern software projects consisting of hundreds or even thousands of components this can not be achieved realistically by manual means.

In order to allow for the verification of the Configuration Management, the code must include configuration information, such as version, revision, date, purpose of the editing session, references to possible software problem report, software change decision, etc.

Rule 2. Use a "Makefile" or its equivalent for building the application.

The process of generating the software from its constituent source files should be as automated as possible. This will reduce possible problems by ensuring that the different parts of the system are always built in a consistent manner by different members of the development team (or even the end customer!)

Rule 3. Write the software to conform to the coding language international standards, i.e. ISO/IEC 9899 and ISO/IEC 14882.

The coding language must conform to its reference definition. In particular,

- C only code should conform to ISO/IEC 9899 Standard.
- C++ code should conform to the ISO/IEC 14882 Standard.

Rule 4. Do not rely on compiler specific features.

Some compilers are capable of handling extensions to the standard language and may also offer additional options such as the automatic initialisation of data variables and memory locations to zero, or the automatic promotion of single precision floating point variables to double precision, etc. The programmer may find it tempting to make use of such extensions and options, but these may simply hide problem areas in the code, which will be exposed when the code is compiled on another hardware platform, or when a different version of the compiler is used.

Rule 5. Use independent tools to provide additional warnings and information about the code.

Compilers are usually only interested in the current module and have no interest in the overall software system. This is usually left to a separate linker. As a result the compiler may accept constructs which are not necessarily portable between modules, such as parameter type mismatch between a function definition in one module and its call in another module.

Compilers are usually tightly coupled to the hardware and operating system upon which they are used. The compiler may therefore accept constructs which are valid

CHAPTER 4
GENERAL PROJECT GUIDELINES

for the current environment but which are not necessarily portable between different hardware platforms. An example of this would be the alignment of data types in memory and the use, or misuse, of pointers to such data types.

Programmers are strongly advised to make use of other tools, such as "lint" for C, which provide additional heuristic checks for problem and non-portable areas of code. Other tools such as those used to extract documentation and cross-reference listings from the code are also useful because they exercise the code in a different way and therefore provide a new angle when looking for potential problems.

Rule 6. Always identify the source of warnings and correct the code to remove them.

The compiler may recognise possible problem areas in the code and issue warning messages. Such code may appear to work correctly but may depend on other factors such as the hardware or compiler being used. If the code is left in this state the impact of the warning message will be lost and at some point in the future will be systematically overlooked. Therefore the code should be examined and corrected so that when warning messages appear the programmer is immediately alerted to any problems.

Rule 7. Do not attempt to optimise the code until it is proved to be necessary.

The code should be designed and written to be as clear as possible so that future programmers will be able to understand and maintain it without the help of the original author. Some programmers feel that they should be writing code which is fast or compact and often have an intuitive feeling for which areas of the code should be optimised for speed or memory purposes. These feelings often prove to be wrong.

By contrast, modern compilers are able to optimise the object code without needing to modify the source code at all. The compiler is usually tightly coupled to the underlying hardware and can make use of its knowledge of the generated object code and hardware to produce more systematic optimisations than the average programmer.

If the optimisation provided by the compiler is not sufficient to improve speed or memory performance then the program must be analysed to determine where improvements should be made. Only when measurement has demonstrated that there is a problem with a particular data structure or algorithm should there be any attempt to rewrite the clear and simple code in order to improve performance.

Rule 8. Provide meaningful comments in the source code and make sure that they are kept up to date.

In "Writing Solid Code" (Appendix A, #16) Maguire argues that the source code is the only component of the overall software project which is guaranteed to be up to date. Therefore the programmer should provide comments in the source code so that

maintenance programmers will be able to understand the code without needing to refer to external documentation.

Rule 9. Enclose debugging code within #ifdef/#endif pairs rather than removing it after use.

The author of the code knows the most appropriate place to add debugging code and which information will be useful for the person who is investigating the code. Therefore any debugging code which the original author provides should be left in place to help maintenance programmers later on. To reduce the impact of the debugging code on the speed or size of the system which is delivered to the customer, this debugging code should be enclosed in #ifdef/#endif pairs so that it is not included in release versions. Different areas or levels of debugging information can be provided by using command line arguments or by setting debug variables within a debugging tool.

```
#ifdef DEBUG
    if (debug == TRUE)
     {
        DebugMouseCoordinates(mouseX, mouseY);
    }
#endif /* DEBUG */
```

Rule 10. Make sure that any code which you use for debugging purposes does not have side effects.

Any debugging code must not affect the overall state of the rest of the system. It may only modify parts of the system which relate to the debugging code itself. It may not modify the rest of the code. The state of the system immediately before and after the section of debugging code must be unchanged. This ensures that the behaviour of the system doesn't change when debugging code is added or removed.

Rule 11. Use appropriate verification tools to ensure that the code conforms to the rules and to catch potential problems as early as possible.

The use of software tools, if available, for the automatic review of code to check for conformance to the rules may help the programmer to find problems.

CHAPTER 5
THE USE OF COMMENTS

Most programmers prefer programming to documenting their code, especially if they are obliged to produce separate documents after the code has been developed.

As mentioned in Maguire's books, "Writing Solid Code" (Appendix A, #16) and "Debugging the Development Process" (Appendix A, #27), the only source of information about the current source code which is guaranteed to be up to date is the source code itself. Therefore programmers should be encouraged to produce code which is as self-documenting as possible. However, it is not always possible to include all information that is needed by using the programming constructs themselves which is why additional comments are needed.

We will return to this issue of self-documenting code when we look at naming conventions.

Rule 12. Write the comments in the common language of the project.

The software project is likely to involve people from various nationalities and language. However, there should be a common language officially used in the project. It happens to be quite often English, but could be another one. The comments included in the code should be produced in the common project language.

Rule 13. Comments in purely C programs must use the /*...*/ delimiters.

In fact C programmers do not have any choice about their comment delimiters.

Rule 14. C++ programs should use the // to end of line convention.

According to the language definition, C++ programmers have a choice of being able to use /*...*/ delimiters or comments which use the // to end of line convention. There are various problems connected with using the /*...*/ style because the comments can't be nested and also because expressions such as "*x/*y" can confuse the pre-processor. It may decide that the expression contains the start of comment resulting in compilation errors at best and an obscure problem to debug at worst.

The examples in this section use /*...*/ comments rather than //... because they illustrate the problems more clearly.

Rule 15. Avoid "fancy-layout" comments because they require time and effort to maintain.

Many programmers like to produce aesthetically pleasing "box comments" such as the example given below, but these require additional effort to keep up to date whenever the text of the comment is changed.

```
/*********************************************************/
/* Title:                                                */
/*    This is a bad style of comment!                    */
/*         --------                                      */
/*        !  Read  !--------→ Store value                */
/*         --------                  |                   */
/*             |                     V                   */
/*             V                  Log(Value)             */
/*        ( Open File )    Compute ----------            */
/*                                    X ← WARNING if 0! */
/* Reason:                                               */
/*    It takes too much effort to maintain when the text */
/*    is changed                                         */
/*********************************************************/
```

If you need an extended comment use a simpler layout such as the counter-example:

```
/*
 * Title:
 *    This is a more manageable comment!
 * Reason:
 *    The programmer doesn't need to spend nearly as much
 *    effort in maintaining the format when the text changes
 */
```

You could argue that in the "more manageable" comment above, the repeated asterisks at the start of line are unnecessary, but they do help to show the extent of the comment. If the programmer makes use of a "Find in Files" type of command or utility the leading asterisk helps separate those fragments of code used for explanatory purposes in a comment from real code.

Note that there is probably a compiler-specific limit to the length of such comments which will not be encountered when using the // to end of line convention because the maximum line length is known (see Rule 18).

Rule 16. Comments must add to the code, not detract from it.

Comments should be used to expand on the information provided by the code, to clarify what is happening, to accurately define the behaviour of the code, to introduce

the specific actions and the coding tricks, to identify possible partial implementation. Avoid comments which obscure the code. The first three examples show poor commenting, and the fourth shows better practices:

```
counter++; /* increment counter */

/* hide the real code */ counter++; /* between comments */

procedureCall(parameter1, /* don't use
              parameter2, but */ parameter3);

/*
 * remove trailing white space from a string using the XYZ
 * algorithm as described in ABC
 */
for (i = strlen(string)-1; i >= 0 && isspace(string[i]); i--)
{
    string[i] = '\0'; /* terminator instead of final space */
}
```

A presentation based on the PDL spirit could be used (input, output, relation, format)

Rule 17. Comments should never be used for "commenting out" code.

If code is no longer required it should be removed. When reviewing code it is never clear whether code which has been commented out is no longer necessary, or has been replaced by other code or is awaiting the completion of some other section of code with which it will interact.

According to Rule 1, the complete source code should be being maintained in a configuration control system so old versions can be retrieved if needed.

Debugging code should be enclosed within #ifdef/#endif pairs (see Rule 9).

CHAPTER 5
THE USE OF COMMENTS

This page is intentionally left blank

CHAPTER 6
GENERAL LAYOUT OF CODE

In the same way that comments affect other more specific areas of the guidelines, code layout is also entwined with other areas. This section provides some guidelines on code formatting and should provide the reader with a feel for the overall importance of the way that code is laid out. The use of white space to provide additional clues of the block structure via consistent indentation is presented, as well as the use of vertical white space. The use of opening and closing braces on lines of their own in a consistent way across all language constructs helps keep the structure of the code clear.

The following rules relating to layout shall be applied to all code unless:

- the code has been produced by a code-generator phase of another tool;
- the code has been provided by a third-party;
- the code is to be integrated with other tools for which a particular layout is necessary or usual.

It is pointless to reformat code which is generated or which comes from a source over which the programmer has little control. Reformatting actually reduces maintainability because it is difficult to see what has changed from one version of the code to the next without spending a lot of effort reformatting it.

6.1 GENERAL

Rule 18. Each project shall define the maximum length in characters of the source code lines (e.g. Each line of source code shall be no more than 80 characters in length).

In this age of graphics workstations and resizable windows this may seem like an old-fashioned restriction harking back to dumb terminals which were limited to 24 rows of 80 characters. This recommendation should be applied so that code can be printed or included in documentation without losing information or the user having to reformat it.

Note that most lines of source code included in this document are less than 65 characters in length even though a smaller font is used than for the ordinary text.

Rule 19. Each nested block of code, including one-line blocks, shall be enclosed within braces.

CHAPTER 6
GENERAL LAYOUT OF CODE

Rule 20. Each project shall define the bracing style to be used (e.g. one of the following: Option 1 - The braces shall appear on separate line and at the same indentation level as the previous block of code Option 2 - The opening brace should appear in the same line of the construct/structure it opens; the closing line should appear at the same indentation level as the previous block of code.).

Enclosing all blocks of code within braces, even if the block consists of only one, possibly empty, statement ensures that the structure of the code is obvious. It also avoids possible problems with the insertion of additional statements in the block, or of using a poorly implemented macro which expands into several statements. The following two contrived examples do not behave in the way that a casual glance might suggest:

```
while (*p++ == 0);
    x = *p;
```

Which is the mistake, the semicolon on the first line, or the indentation of the second?

```
#define SWAP(x, y)  tmp = x; x = y; y = tmp;

if (x < y)
    SWAP(x,y);
```

The body of the "if" statement only really contains "tmp = x;" and the other two statements hidden within the SWAP macro are always executed. For details of macros see Rule 111 and Rule 112.

Having the opening brace on a line on its own (rather than the "traditional K&R style") provides a more visible separation between the logic of control statements and their contents, as can be seen in the following sections.

Rule 21. Each project shall define its own indentation rules (e.g. The contents of each nested block shall be indented by 4 spaces compared to the braces which delimit the block).

The standard "hard-tab" character would provide indentation of eight spaces but this would give lines which reached the 80 character line limit very quickly and result in breaking lines unnecessarily. Four spaces allow for clear indentation without lines reaching the right margin too quickly. Functions which contain too many levels of

indentation and whose lines need to be broken at 80 characters should probably be split into smaller routines.

6.2 LAYOUT OF "IF" STATEMENTS

```
if (expression)
{
    statement(s);
}
```

```
if (expression) {
        statement(s);
            }
```

The "if-else" statement continues the idea that opening and closing braces for blocks should exist on lines of their own.

```
if (expression)
{
    statement(s);
}
else
{
    statement(s);
}
```

```
if (expression) {
        statmemnt(s);
            }
        else {
        statement(s);
            }
```

The layout of "else-if" constructs is one, which can only be justified by breaking other rules concerning layout. On the one hand the trailing "if" statement is really a single statement block and according to Rule 19 should be enclosed in braces. However this is not really practical because it means that a long sequence of "if-then-else" constructs will march off the right of the page. Supporters of "else-if" as a construct in its own right would argue that the "if" should follow immediately after the "else" and on the same line. However much can be said for the symmetry and clarity of having all "if" statements of such a sequence at the same relative position on the line

```
if (expression)
{
    statement(s);
}
else
if (expression)
{
    statement(s);
}
```

```
if (expression) {
    statement(s);
} else if (expression) {
    statement(s);

}
```

More complicated logic, which would otherwise extend the line beyond its permitted length can be written as given in the following example which also illustrates the value of the opening brace as a vertical separator between the expressions and the statements:

```
if (expression1 &&
    expression2)
{
    statement(s);
}
```

```
if (expression1 &&
          expression2) {
          statement(s);
                   }
```

6.3 LAYOUT OF "SWITCH" STATEMENTS

```
switch (expression)
{
    case 0:
        statement(s);
        break;

    case 1:
        statement(s);
        /* no break so FALLS THROUGH */

    case 2:
    case 3:
        statement(s);
        break;

    default:
        statement(s);
        break;
}
```

Rule 22. Each case within a switch statement must contain a break statement or a "fall-through" comment.

Where one case within a switch statement performs some pre-processing before falling through to the code of the following case this must be commented in the code. See case 1 in the example above.

Where a case shares all of its code with the following case this does not need to be commented because this is immediately apparent. See cases 2 and 3 in the example.

All other cases, including the last, must have an explicit break statement.

Explicit break statements and fall-through comments indicate those places in the code where it is safe, or unsafe, for a programmer to add an additional case statement without interfering with the existing code.

Rule 23. All switch statements shall have a default: clause.

All switch statements shall have a default case even if the programmer believes that the default case can never be reached. This allows for possible errors in the code, or

CHAPTER 6
GENERAL LAYOUT OF CODE

corruption of data, to be detected more easily.

6.4 LAYOUT OF "FOR" STATEMENTS

```
for (expression1; expression2; expression3)
{
    statement(s);
}
```

```
for (expression1; expression2; expression3) {
                statement(s);
            }
```

More complicated logic, which would otherwise extend the line beyond its permitted length can be written as given in the following example which also illustrates the value of the opening brace as a vertical separator between the expressions and the statements:

```
for (expression1;
     expression2;
     expression3)
{
    statement(s);
}
```

```
for (expression1;
     expression2;
     expression3) {
    statement(s);
}
```

6.5 LAYOUT OF "WHILE" STATEMENTS

```
while (expression)
{
    statement(s);
}
```

```
while (expression) {
    statement(s);
}
```

More complicated logic, which would otherwise extend the line beyond its permitted length can be written as given in the following example which also illustrates the value of the opening brace as a vertical separator between the expressions and the statements:

```
while (expression &&
       expression)
{
    statement(s);
}
```

6.6 LAYOUT OF "DO-WHILE" STATEMENTS

```
do
{
    statement(s);
}
while (expression);
```

```
do {
    statement(s);
} while (expression);
```

More complicated logic, which would otherwise extend the line beyond its permitted length can be written as given in the following example which also illustrates the value of the opening brace as a vertical separator between the expressions and the statements:

```
do
{
    statement(s);
}
while (expression &&
       expression);
```

CHAPTER 7
THE OVERALL SOFTWARE STRUCTURE

7.1 SUBSYSTEMS

A subsystem consists of the source code and associated files, which are used to implement a free-standing part of the overall software system. A subsystem may use libraries and modules, which are also used in other subsystems.

7.2 LIBRARIES

A library consists of a collection of modules, which define a set of inter-working abstract data types (ADTs) and associated functions. For software written using C++, a library is also likely to contain classes.

Rule 24. Use separate directories for each of the subsystems and libraries defined during the design phase.

The design phase should have broken the overall software system into logical subsystems and libraries as a means of managing the complexity of the overall system. In order to reinforce the logical division of hundreds, if not thousands of modules into these subsystems and libraries the modules should also be divided physically. The easiest way of doing this is to dedicate a directory to each of the subsystems and libraries.

Each library should really be treated as a "black box" which only exposes its interface to the programmers who use it. By keeping the source code of the library separate from the code which uses it the "black box" nature of the library is reinforced. If the programmer using the library does not have easy access to the source code for the library then he will be more likely to conform to the library's published interface and not make use of any knowledge of the internal workings of the library.

7.3 MODULES

A module is a conceptual unit which consists of a set of files which together make up the interface and implementation of a particular abstract data type (ADT) or class

(C++ only), or for a logical grouping of functions.

Rule 25. Use a utility or proforma to provide the starting point for all files within each particular subsystem, library or module.

In order to ease understanding and maintenance, similar types of files should have a similar "look and feel". In a project involving more than one programmer this can be achieved easily by providing a standard starting point for each type of file. The simplest way of enforcing this is to provide a proforma for each type of file that contains the standard constructs defined by these rules and the project. The programmer can use this proforma as the foundation for further work. Each subsystem or library may need its own set of proforma files but these can also be based on a project wide standard.

Another possibility is to provide a utility program or script, which can generate a file which contains the standard constructs. Such a tool could also prompt the user for information to be inserted into the file and also incorporate information about the particular subsystem, library or module. The programmer will not face the immediate task of customising the basic file before work begins and therefore will not be tempted to skip what might be considered to be a chore. All files are more likely to contain relevant information.

7.4 FILES

The term ".h file" is used in this document to refer to the public interface file because of the long-standing Unix/C practice of using a ".h" extension to the file name, although other operating systems and compilers may require a different extension. The term ".c file" is used in this document to refer to an implementation file for similar historical reasons.

Rule 26. Source code shall be separated into a public interface (the ".h" file) and a private implementation (the ".c" file).

The interface file provides a single point of declaration of the data entities and functions provided by a particular module. The interface file should be #included in all modules which refer to or make use of the data entities and functions provided. Sections of the interface file must not be copied into other modules!

Although this document refers to a single implementation file there may be reasons for breaking the implementation of a particular module into several ".c" files such as a limit on the size of an individual file.

Rule 27. The interface file must be included in its own corresponding implementation file.

Including the interface file for a module into its corresponding implementation file means that the compiler can check that they are consistent.

Rule 28. Each file shall contain a standard comment header block.

In the absence of other documentation the files themselves should contain enough information for the programmer to use them. The standard header comment block in the interface file should contain information on what the module does whereas the implementation file contains information on how it does it.

Using a standard header comment block such as the one shown in Appendix B means that the information is always provided in a standard manner. This can be useful if documentation is being generated from the code itself.

The standard header comment block consists of a series of keywords and associated text. Therefore the keywords used in the two types of file may be different. The use of keywords provides a degree of flexibility as new keywords can be added to suit the individual application.

The interface file may be the only part of a module, which is available to a programmer using that module because the source code for the implementation of the module is unavailable. This is quite common when the programming team is split into different groups where one group may supply the interface files and an object library to the other groups without making the complete source code for the library available.

Rule 29. The public interface file shall be self-contained and self-consistent.

It is not uncommon for one module to be built on top of other modules, which may in turn depend upon yet more modules, and so on. The interface file should #include the interface files of the other modules upon which it depends directly. They of course will #include the lower level modules. This means that any module only needs to #include those interface files for which there is a direct dependence. This reduces the number of interface files which need to be explicitly #included in each module. This therefore reduces the number of modules, which need to be modified if a lower level module is changed.

The interface file must not #include files, which are not strictly needed for the interface. This leads to unnecessary dependencies between the files, which will affect compilation. Individual modules will take longer to compile if they #include more files than needed and more modules are likely to be recompiled than necessary.

For example a Window is defined only in terms of Rectangles, which are in turn defined in terms of Points. The Window.h file should only need to #include the Rectangle.h file because the Rectangle.h file #includes the Point.h file. The user of Window will only need to #include Window.h rather than having to #include Point.h, Rectangle.h and

Window.h. If Rectangle is subsequently changed to use Lines rather than Points, the user of Window may not need to modify the Window module.

There is another way of reducing the interdependency of files. If a particular file only contains pointers to a particular class or structure rather than complete instances of that class or structure then the file may only need to have a forward-declaration rather than #include the interface file. In the example above, if a Window only contains pointers to Rectangles, then Window.h may only need a forward declaration for Rectangle rather than including the complete Rectangle.h interface file.

Rule 30. The contents of the interface file shall be surrounded by #ifdef/#endif preprocessor directives in order to avoid problems of multiple inclusion.

The previous rule leads to the possibility that a module may indirectly #include an interface file more than once. In an ideal world each interface file would be constructed in such a way that this would not make any difference, but unfortunately this is not an ideal world. Compilers tend to complain about duplicate data and type declarations. Some constructs involving the pre-processor can only really be applied once. Even if these things could be handled, it does not really make sense for the compiler to reprocess each file every time it is encountered. Therefore to reduce semantic and speed problems of #including interface files more than once, the contents of each interface file should be enclosed in a "guard" directive to the pre-processor. For example, consider the interface file for the Window module in the User Interface library:

```
/*
 * Window.h
 */

#ifndef Ui_Window_h
#define Ui_Window_h 1

#include "Rectangle.h"

typedef struct
{
    char *name;
    Rectangle rect;
} Window;

#endif /* Ui_Window_h */
```

Note that the pre-processor variable which is used should correspond to the name of the module or interface file in an obvious way and should also reflect the subsystem or library to which the module belongs. Characters which are not valid in pre-processor variables should be replaced by an underscore.

Rule 31. The interface file shall contain declarations only.

The interface file may only contain declarations of data entities such as variables and functions and not their definitions, i.e. they must be declared as being "`extern`". The interface file contains information that such things are defined somewhere in the source code but does not know where.

The variables and functions must be defined in the corresponding implementation file, which must #include the interface file in order to provide consistency checking.

If data entities were to be defined in the interface file then every module which included the interface file would have its own version of the data entities. For duplicate variables this can lead to problems where a value is assigned to a variable in one module but this does not correspond to the variable used in another module so the value which is used is incorrect. At the very least duplicate functions mean increased object size and unnecessary redundancy. If the function contains a static data variable then each version will have its own separate variable with the problems outlined above. Problems involving duplicate variables can be very hard to isolate.

CHAPTER 7
THE OVERALL SOFTWARE STRUCTURE

This page is intentionally left blank

CHAPTER 8
NAMING

C has traditionally been a "terse" language, partly because of the limitations of the original suites of compiler tools. This terseness has been continued in most of the text books and publicly available source code. However most modern compilers allow longer and more meaningful names and modern software engineering practices encourage the use of descriptive names.

In "Writing Solid Code" Maguire emphasises that the source code is the only part of the overall software product which is guaranteed to be up to date, especially during periods of high development activity if code is produced first and documented at a later date. The source code may therefore be more up to date than any documentation, which describes it. This means that the source code should be as clear and self-documenting as possible. One obvious means of achieving this is to use descriptive names within the source code.

This chapter is concerned with issues of naming within the source code. It does not deal with the naming of files and directories because there may be constraints imposed by the operating system such as the length of the name or the use of upper/lower case letters and which are therefore beyond the scope of this document.

It is worth bearing in mind that the naming of entities in C is more restrictive than in C++ because all C entities must have unique names. In C++ members from several different classes may have the same name because the combination of class and member name will be unique.

When considering the names of entities it is worth remembering that there are three possible levels of access for each entity:

- System global entities: if something is declared so that it is visible outside its own module then it is also visible anywhere in the code even if it is only needed in one other module.

- Module global entities: if something is declared at the file level within a module, i.e. outside an individual function, then it is also visible throughout that module.

- Local entities: if something is declared within a function or block then it is only visible within that function or block.

8.1 GENERAL NAMING CONVENTIONS

Rule 32. Names shall not start with an underscore character (_).

Although C and C++ allow identifiers which begin with an underscore the programmer is strongly advised against using such names. The [emerging] C++ standard specifies that identifiers, which begin with two underscores, are reserved for compiler and library use. Traditional C libraries make use of identifiers which begin with an underscore for low level implementation details which the programmer is never supposed to use directly. Therefore to avoid any inadvertent conflict with compiler, library or operating system identifiers the programmer is expressly forbidden to use names which begin with an underscore.

Rule 33. Names shall be meaningful and consistent.

Descriptive names are preferable to "random" names. However the descriptive name must reflect the use of the entity over its lifetime. For example "`count`" is better than "`xyz`" for the name of a variable which contains the total number of something but not if it is used to hold an error code.

Rule 34. Names containing abbreviations should be considered carefully to avoid ambiguity.

In the absence of a list of standard abbreviations different programmers are likely to choose different abbreviations to represent a particular word or concept. This is especially true in the multi-lingual environment found in many ESA projects. Abbreviations should be used with care. For example does "`opts`" refer to "options" or the "old points" or something else?

Rule 35. Avoid using similar names, which may be easily confused or mistyped.

Names which differ by only one character, especially if the difference is between "`1`" (the digit) and "`l`" (the letter) or "`0`" (the digit) and "`O`" (the letter). Avoid names, which consist of similar meaning words because it is easy to confuse them. Consider the differences between "`x1`" and "`xl`" and between "`countX`" and "`numberX`".

Rule 36. Names for system global entities shall contain a prefix, which denotes which subsystem or library contains the definition of that entity.

The subsystem or library prefix provides additional information about the origin of an entity, which can be useful to the programmer. The basic name (without the prefix) can be reused elsewhere in the source code without causing a name clash.

Rule 37. Names, which have wide scope and long lifetimes, should be longer than names with narrow scope and short lifetimes.

Names for entities which are local to a block or function may be short, e.g. `count`, `result`. Names for entities which are available throughout a particular module should be longer, e.g. `maxNumber`, `startPoint`. Globally visible entities should have long descriptive names, including library prefix, so that their origin and use are clear, e.g. `uiWindowIndex`, `uiMousePosition`.

Rule 38. Each project shall define its own specific rules for naming conventions. The following rules in this chapter are provided as example.

8.2 PRE-PROCESSOR NAMES

In this and the following sections, "word" is also used to mean an abbreviated word, such as "max" and acronym type abbreviations such as "RGB".

Rule 39. Pre-processor names shall consist of upper case words separated by underscore.

Established C convention uses upper case only names for pre-processor objects such as literal constants and macros. This emphasises the difference between true C constructs and pre-processor entities, which are not really part of the C language at all.

Pre-processor entities provide a substitution mechanism in the code itself rather than being part of the code. Pre-processor names are not bound by any of the rules of scope which apply to C and C++ names and exist from their definition until the end of compilation of the module unless explicitly redefined or even undefined.

Programmers are therefore advised to use only upper case in order to distinguish pre-processor entities from true C/C++ constructs. The use of underscore to separate words improves the readability of the name.

Pre-processor names which are defined in an interface file are not truly global in the sense that has been described previously but are local copies within any module which uses that interface file. It is still helpful to be able to determine where such names are defined. Therefore "global" pre-processor names should also include the prefix corresponding to the subsystem or library in which they are defined, e.g. `UI_MAX_WINDOW_INDEX`. "Local" pre-processor names do not need this additional prefix, e.g. `SUCCESSFUL`.

8.3 VARIABLE NAMES

Rule 40. Variable names consist of one or more words where each word except the first is capitalised.

This scheme allows the use of simple, single word identifiers for local variables such as loop counters and at the same time provides clarity for longer, multi-word names without the lengthening effect of using underscore as a word separator. Note that the library prefix included in global names appears in lower case.

As a matter of good programming style, variable names should also reflect some invariant property of the variable. A variable called "`currentLine`" should never be used to refer to "`previousLine`" and if there can ever be more than "`maxNumLines`" then it is clearly not an appropriate name.

Acronym type abbreviations are still subject to the "capitalisation scheme" because this allows for clearer separation of words within the name and makes them easier to read. For example, "`maxRgbLevel`" is preferable to "`maxRGBLevel`" (or even "`maxRGB_Level`") because the words are more easily distinguished without any one word being dominant.

Examples of variable names: `i`, `length`, `oldWidth`, `uiMouseDeltaX`.

8.4 FUNCTION NAMES

Rule 41. Function names consist of one or more words where each word is capitalised.

Capitalising the first word enables the user to distinguish between variables and functions with similar names. Capitalising subsequent words provides clarity without the increase in length associated with underscores. Note that the library prefix included in global function names is also capitalised.

Function names should probably use a combination of verb(s) and noun(s) although care must be take to avoid ambiguity. For example, does

```
if (EmptyThing(&thing) == TRUE)
```

mean that the variable "`thing`" is in fact empty, or that it has been emptied successfully? Changing the name to "`IsEmptyThing`()" would make the meaning clear.

Examples of function names: `IsValid()`, `SetDefaults()`, `UiShowInputForm()`.

8.5 TYPE AND CLASS NAMES

Rule 42. User defined type and class names consist of one or more words where each word is capitalised plus an appropriate prefix or suffix.

Using a capital letter for each word and an appropriate prefix or suffix as defined in the following rules distinguishes these names from both variables and functions. Note that names should first include a library prefix if necessary.

Rule 43. User defined type names shall begin with "T" or end with "_t" or "_type".

Rule 44. Class names shall begin with "C" or end with "_c" or "_class".

In traditional C programs various system libraries use "`typedef`" to define useful types. These types have names ending in "`_t`" to discriminate them from ordinary variable names. ANSI C and the [emerging] C++ standard actually define a series of such names, which are used by standard library routines. They are not strictly speaking part of the language itself but are provided for convenience and portability reasons. Examples include "`size_t`" and "`ptrdiff_t`".

If the same rationale were applied to the naming of classes then all class names should have a "`_c`" appended to the name. However, one emerging de facto standard within the C++ community is the use of an initial "`C`" to denote class names. If the reverse reasoning is applied to "`typedef`" names then type names should be prefixed with a "`T`".

Supporters of consistency argue that if a programmer uses "`_t`" then it is logical to use "`_c`" (or "`C`" and "`T`"). Others maintain that typedef names should not use "`_t`" so that user defined types are distinct from the those supplied by the system. Similar arguments apply to the use of "`C`" as a prefix for a class name. As global names should have a library prefix and other names have more limited scope these are not really strong arguments.

There is no easy middle ground, which will keep everyone happy. It is rather difficult to allow both consistency with existing library usage on the one hand and distinctness from the same libraries on the other hand. Therefore this document allows a limited amount of flexibility in this area, although whichever option is chosen should be used consistently throughout the source code.

Note that in C a programmer must explicitly associate a type name with "`struct`", "`union`" and "`enum`" constructs using "`typedef`". In C++ the use of

"`typedef`" to define type names for such constructs is not necessary because the compiler does this automatically.

8.6 CLASS MEMBER VARIABLE NAMES

Classes are not like traditional C structures where all member names are obvious from the declaration of the structure and where the structure and member names are always used together. Class member names are often used on their own because the current object name is understood (i.e. via the `this` pointer).

If one class is derived from another it is possible for it to use members from the base class without them being immediately obvious in the derived class's own declaration. As such they can appear without an obvious declaration in a member function and therefore resemble ordinary variables with wider scope. This can be confusing.

The C++ community as a whole recognises the need to differentiate between the names of class member variables and "ordinary" variables. Some people use a leading underscore to denote member variables, while others use a trailing underscore. However, this standard has already ruled against the use of names beginning with underscores, and trailing underscores are not particularly noticeable. Other people use a "`my`" or "`their`" prefix for member variables, but the scheme which appears to be gaining popularity is the use of a single letter followed by an underscore as a prefix to a variable name. Various letters are used to denote different properties of the variable. The annotations, which deal with ordinary variables, have already been addressed by other means in this document. The two prefixes, which deal with class member variables, are concise and will therefore be adopted by these guidelines.

8.7 ENUMERATED TYPES

Rule 45. Each enumeration within an enumerated type shall have a consistent prefix.

It is not uncommon for different enumerated types to be used within a program which have similar characteristics, such as those representing the "status" of different parts of the system. Although they are distinct, enumerations within one type may have similar roles to those of another type. To reduce the risk of name clashes for such enumerations, the name of each enumeration should include a prefix which denotes the exact enumerated type to which it belongs. This also reduces possible confusion when converting to or from integer variables.

```
typedef enum
{
    ValveUnknown,   // the status of the valve is unknown
```

```
    ValveOpen,      // the valve is open
    ValveClosed     // the valve is closed
} ValveStatus_t;
```

8.8 SAMPLE CODE

Consider an example of a project containing a library, which deals with access to a remote database. The code as presented here is obviously incomplete but should illustrate the principles behind the naming conventions, which have been described above.

Some useful constants and types might be declared using:

```
#define DB_MAX_INDEX 9999              /* for C */
const unsigned int DbMaxIndex = 9999;  // for C++

typedef unsigned int DbIndex_t;
typedef enum
{
    DbError,     // an error occurred while accessing the
database
    DbOpen,     // the database was opened successfully
    DbClose     // the database is now closed
} DbStatus_t;
```

One particular module might define a global hash function for use with local caching of database entries to reduce network delays:

```
unsigned short DbHashIndex(
    const DbIndex_t index
)
{
    // example function only - not for real use!

    unsigned short hash;

    hash = (index % 99) + 1;

    return(hash);
}
```

Another module might be concerned with tables in the database and contain:

```
class DbTable_c
{
    // member declarations
};
```

This class might contain a member variable which is used by all of the table objects:

```
static DbStatus_t s_currentStatus;
```

and a member variable containing the number of rows in the table which have been read from the remote database and saved in the local cache:

```
unsigned int m_NumCachedRows;
```

as well as a member function which returns the total number of rows for the real table which is held in the remote database and another which returns the number of cached rows:

```
unsigned int GetNumRows();
unsigned int GetNumCachedRows();
```

CHAPTER 9
DECLARATIONS

9.1 GENERAL DECLARATION GUIDELINES

Rule 46. Each declaration should start on its own line and have an explanatory comment.

For some language constructs, such as the definition of pre-processor macros, this rule is implicit, and for others such as function or class definition it is unlikely that a programmer would want to declare more than one per line. However it provides a general principle for all constructs and should be explicitly applied for members of enumeration, structures, classes, variable declarations and function parameters.

The rule means that the programmer always has the opportunity of adding or removing declaration lines without having to modify other declarations and provides space at the end of the line for an explanatory comment.

Rule 47. Entities should be declared to have the shortest lifetime or most limited scope that is reasonable.

The namespace within the source code is limited and therefore it makes sense not to clutter it up unnecessarily. Some entities correspond to the internal implementation of particular modules and should not be exposed elsewhere.

Rule 48. Global entities must be declared in the interface file for the module.

The only way to ensure consistency between different modules is to use the same information in each module. The interface file forms that single point of reference.

Rule 49. Declarations of "extern" variables and functions may only appear in interface files.

An interface file provides a means of telling modules that a variable or function has been defined somewhere but not exactly where. If the programmer uses "`extern`" declarations within implementation files this increases the number of places which need to be changed if the variable or function is changed.

Rule 50. Declarations of static or automatic variables and functions may only appear in implementation files.

This rule follows on from the previous rule but it is, in fact, poorly worded. A variable, which is declared as “extern” only, provides a placeholder, which is used to bind

the variable name with the true definition of the variable.

A static or automatic variable declaration really provides the direct declaration of the variable. Therefore each module which contains such a declaration contains a definition of its own variable. In many cases this is perfectly acceptable and allows variable names to be reused for different purposes in different modules.

If the declaration of a static or automatic variable appears in an interface file, then each module which includes the interface file will contain its own definition of its own variable. This is likely to lead to confusion and errors which are difficult to track down because the programmer will assume that the variable has global scope when in fact there are many different versions of the variable. See also Rule 31.

Rule 51. Declarations should appear in the order: constants and macros; types, structs and classes; variables; and functions.

Symbolic constants and macros are likely to be used within type, struct and class declarations, which in turn are likely to be used for declaring variables. Such system or file global variables are likely to be used in the functions. Grouping each category of entities together allows the programmer to find a particular declaration much more quickly than if entities were declared "as needed".

9.2 DECLARATION OF CONSTANTS

Rule 52. Symbolic constants shall be used in the code. "Magic" numbers and strings are expressly forbidden.

Code, which contains literal constants, can be very hard to understand because there may be no obvious reason why a particular number, size or string appears in the code. This makes maintenance of the code difficult. Maintenance or future modification of the code is also complicated because whenever one of these literal constants needs to be changed the programmer must examine every occurrence of that literal constant in the code to determine whether it is being used for the same purpose before being able to update it.

It is far better to define a symbolic constant once for a particular meaning of a literal constant and then use this symbolic constant. This improves readability of the code in the areas where the symbolic constant is used, and distinguishes between different symbolic constants, which may have the same literal constant value. Modification is also easier because there is only one place, which needs to be changed.

Rule 53. For C: all symbolic constants shall be provided using an "enum" or the #define mechanism.

The use of an "enum" is preferred over the use of #define for small groups of related integer constants because the compiler is able to perform additional type checking.

Rule 54. For C++: all symbolic constants shall be provided using an "enum" or by using the "const" keyword.

C++ programmers should avoid the use of #define wherever possible because there are potential problems with the scope - or rather the lack of scope - of preprocessor variables. C++ programmers can declare variables as being "const" and use these in place of #define.

9.3 DECLARATION OF VARIABLES

Global variables shall be declared as "extern" towards the top of the interface file, and defined at the top of the implementation file, above any function declarations. The definition in the implementation file may also include initialisation of the variable.

Rule 55. Each variable shall have its own personal declaration, on its own line.

Having a complete variable declaration on each line means that the type and name and possibly a brief descriptive comment about use or properties of that variable are always next to each other.

One problem with variable declarations is that each variable has a type and may also have a "type-qualifier". Some people believe that the "type-qualifier" belongs with the type, and others believe that it belongs with the variable. Where variables with different "type-qualifiers" are declared in lists it is not always possible to position the "type-qualifiers" in a consistent manner. With one variable declaration per line there can be no confusion.

Whichever style is chosen should be used consistently throughout the code!

In the absence of any logical grouping of individual variables, the variable declarations should be grouped together by type and alphabetically within each type. This allows declarations to be found more easily.

```
/*
 * these declarations are unclear
 */
int x, y, *z;       /* x,y are ints but z is pointer to int
*/
char* reply, c;     /* reply is pointer to char, but c is
char! */
```

```
/*
 * these declarations are better
 */
char c;                /* current character during string
handling */
char *reply;       /* user's response to prompt */

int x;             /* x coordinate within frame */
int y;             /* y coordinate within frame */
int *z;            /* pointer to z coordinate */

/*
 * these declarations using alternative form
 */
char* reply;       /* user's response to prompt */
int* z;            /* pointer to z coordinate */
```

9.4 DECLARATION OF TYPES

Programmers should be encouraged to use "typedef" to create type names which are meaningful in the problem domain. This promotes readability because it is immediately obvious what the "dimensions" of a variable should be.

9.5 DECLARATION OF ENUMS

We can apply Rule 19, Rule 43 and Rule 45 to give the following general layout for the declaration of an enumerated type:

```
typedef enum
{
    ValveUnknown,   // the status of the valve is unknown
    ValveOpen,      // the valve is open
    ValveClosed     // the valve is closed
} ValveStatus_t;
```

The compiler will number the members from zero if the programmer does not specify particular values.

Note that in C++ where simple variables are initialised to zero it can be useful for the first member of an enumeration to correspond to an unknown, warning or error condition. If the programmer fails to initialise the enum variable correctly this default value should trigger warning code later in the program.

CHAPTER 9
DECLARATIONS

This page is intentionally left blank

CHAPTER 10
STRUCTS AND UNIONS

Rule 56. [C++ only] Structs should be converted to explicit classes where possible.

In C++ a struct is the equivalent of a class whose members are all public. Because it is effectively a class, it can have constructors, a destructor, assignment operators and other member functions.

In C a struct may contain member variables only. It may not contain member functions, although a data member variable may be a pointer to a function.

This may seem like a minor difference, but it means that the programmer must be careful when converting a C struct into a C++ struct because the C++ compiler will generate various member functions implicitly if the user does not supply them. It may not be possible when calling C routines which expect structs, but converting structs (i.e. implicit classes) to be explicit classes forces the programmer to provide the functions which the compiler may otherwise supply.

Please see Chapter 13: "CLASSES [C++ ONLY]."

According to "The C++ Programming Language (Second Edition)" a union can be thought of as a structure whose member objects all begin at offset zero and which may have member functions but not virtual member functions. Unions also have other limitations, but the description above demonstrates that just as there is a difference between structs in C and C++ so there is also a difference in unions.

Unions are used much less frequently in C++ than they are in C. The common use of a structure containing some type identifier followed by a union of the types can be accomplished by using suitable class hierarchies and using the polymorphism designed into the C++ language.

We can apply Rule 19, Rule 20, Rule 43 and Rule 46 to give the following general layout for a union and structure:

```
typedef enum
{
    ShapeUnknown,            /* zero corresponds to invalid
shape */
    ShapeSquare,
    ShapeCircle
} ShapeId_t;
```

```
typedef struct
{
    ShapeId_t  id;        /* which shape is held in union */
    union
    {
        Square_t square;
        Circle_t circle;
    } shape;
} Shape_t;
```

Rule 57. A self-referential struct must have a tag name or forward declaration.

The use of a tag name may be clearer in that the complete declaration appears in one place, but means that two names are used for each struct. Note that the tag name is only ever used in the main declaration of the struct.

```
typedef struct BinaryTree
{
    int nValue;                          /* useful info */
    struct BinaryTree *pLeftSubTree;     /* values < nValue */
    struct BinaryTree *pRightSubTree;    /* values >= nValue */
} BinaryTree_t;
```

The use of a forward declaration is less clear, but only uses one name. The main declaration of the struct resembles the C++ approach even when using C so this can provide an upgrade path for the code without major modification. If the internal details of the struct can be hidden within the implementation file then only the forward declaration will be needed in the interface file (see Rule 29).

```
typedef struct BinaryTree_t BinaryTree_t;

struct BinaryTree_t
{
    int nValue;                          /* useful info */
    BinaryTree_t *pLeftSubTree;          /* values < nValue */
    BinaryTree_t *pRightSubTree;         /* values >= nValue */
};
```

CHAPTER 11
GUIDELINES FOR FUNCTIONS

Rule 58. The "traditional K&R style" of forward-declaration of functions is expressly forbidden.

The "old K & R style" of simply declaring the function name as "extern" with an optional return type was particularly error prone because the compiler assumed that the function returned an integer if no return type was specified. There was also no cross-checking between the actual declaration and the function call of the number or type of any parameters used.

Rule 59. A full function prototype shall be declared in the interface file(s) for each globally available function.

The ANSI C and C++ standard provide function prototypes as a means of "forward-declarations" of function, which are defined elsewhere. Function prototypes are a big improvement on the "old K & R style" because the compiler is able to provide consistency checking of the return type and the number and type of arguments given by the prototype compared to those used in the function call or the actual declaration of the function.

Rule 60. Each function shall have an explanatory header comment.

Each function (or prototype) shall be preceded by a brief comment block, which describes the purpose of the function and its parameters. Any assumptions made about the parameters should also be mentioned. See also Rule 64 and Rule 65.

Rule 61. Each function shall have an explicit return type. A function, which returns no value, shall be declared as returning "void".

A function, which does not have an explicit return type, is assumed to return an integer. This can lead to problems if there is a mismatch between what the caller of the function expects and what the function actually returns. If the return type is not specified explicitly the compiler cannot issue appropriate warnings if the function is called incorrectly or if the return value is invalid.

Rule 62. A function, which takes no parameters, shall be declared with a "void" argument. [C only]

If a function does not take any arguments then the programmer should make this

explicit by specifying "void" in the argument list. This should also be reflected in the function prototype. This avoids any confusion between the function prototype and the "old K & R style" of declaration, which contained no information about the function's arguments.

C++ is stricter in its type checking and function signature matching than C so there is not the same possibility of confusion. The [emerging] C++ standard allows both an empty argument list and the traditional C "void" argument list.

Rule 63. A parameter, which is not changed by the function, should be declared "const".

The compiler should be used to trap as many problem areas as possible. If a parameter should remain unchanged by a function then the programmer should inform the compiler so that it can issue a warning if the parameter is changed inadvertently.

Rule 64. The "assert()" macro, or similar, should be used to validate function parameters during program development.

The ANSI C and [emerging] C++ standard provide an "assert()" macro. The "assert()" macro causes the program to abort with some diagnostic information if the integer expression which is given as the argument evaluates to zero, i.e. false. This facility is enabled during development and disabled prior to release by defining a particular pre-processor variable before compilation.

During development it is common for a parameter to contain a value which is incorrect. There may be several reasons for this: there is an error in the code or the value was not foreseen or the code, which handles this particular value, is not yet available. The use of assertions to validate the parameters means that such values are trapped the first time that they are encountered during development.

Rule 65. Each function shall be defined using the following layout:

```
/*
 * FunctionName() returns the XYZ coefficient using parameter1
 * and parameter2 for the boundary conditions. Assumes that both
 * parameter1 and parameter2 are valid.
 */

ReturnType FunctionName(
    ParamType1 parameter1,  /* comment on parameter1 */
    ParamType2 parameter2   /* comment on parameter2 */
)
{
    /* local declarations */
```

```
    assert( IsValidParam1(parameter1) == True );
    assert( IsValidParam2(parameter2) == True );

    /* body of function */
}
```

Rule 66. A function may not return a reference or pointer to one of its own local automatic variables.

With the notable exception of static data variables within a function, all local variables are created on entry into the function and returned to the system on exit from the function. Therefore the caller has no guarantee that the memory corresponding to what was once the local data variable still contains anything useful or whether it has already been overwritten by something else.

The programmer should be careful not to introduce other potential memory problems in an attempt to work around the one described here. Although it is unlikely to occur in an ordinary function which return simple types, please see Rule 83 for further details.

CHAPTER 11
GUIDELINES FOR FUNCTIONS

This page is intentionally left blank

CHAPTER 12
CLASSES [C++ ONLY]

12.1 GENERAL CLASS GUIDELINES

Rule 67. Class declarations shall contain the "public:", "protected:" and "private:" keywords exactly once.

```
class CExample
{
    public:
        /* public member functions, if any */

    protected:
        /* protected member variables and functions, if any */

    private:
        /* private member variables and functions, if any */
};
```

Rule 68. Class member variables must not be declared "public".

One of the frequently quoted benefits of Object Oriented systems is the ability to encapsulate data within an object and hide the implementation details behind a strictly enforced interface.

If the class member variables are hidden behind more widely available accessor functions then the underlying implementation of the class can be hidden from the user of the class. The implementation may change without effecting the interface itself. This means that other areas of code do not need to change simply because the internal details of a class change. If member variables were public, and the programmer used the directly elsewhere in the code then all this code must be modified and recompiled if the member variable change.

The classic example, which is always given, is a class representing complex numbers, which offers accessor routines for the real and imaginary parts of the complex number, as well as accessor routines for the modulus and argument of the complex number. The user of the class does not need to know which representation is used internally.

If the member variables are hidden, any interaction between the members

variables can be strictly controlled within the class itself so they are guaranteed to be in a consistent state, thereby reducing the amount of checking code which is needed before their values are used elsewhere.

```
class CRange
{
    public:
        // …
        void SetLimits(int lowerLimit, int upper Limit);

    private:
        int m_loweLimit;    // m_lowerLimit <= m_upperLimit
        int m_upperLimit;
};

void CRange::SetLimits(
    int lowerLimit,
    int upperLimit
)
{
    assert(lowerLimit <= upperLimit);

    if (lowerLimit <= upperLimit)
    {
        m_lowerLimit = lowerLimit;
        m_upper Limit = upperLimit;
    } else
    {
        m_lowerLimit = upperLimit;
        m_upper Limit = lowerLimit;
    }
}
```

In the example above the programmer who uses this class does not have access to the individual member variables as is obliged to use the `SetLimits()` function which guarantees that the limits are consistent.

Note that during development the "`assert()`" macro is used to check that the parameters are passed correctly, i.e. that there is no programming error in the caller. In the release version of the code, when the "assert()" macro becomes a no-op the incorrect parameters are still handled correctly. See Rule 64 and Rule 105.

12.2 CONSTRUCTORS AND DESTRUCTORS

If the programmer does not supply these member functions explicitly the compiler will provide implicit versions. These implicit version provide "bitwise" initialisation or copying which are not always appropriate in all circumstances particularly if the class

contains pointer to dynamic memory. Some people would argue that these member functions should only be provided explicitly when the class needs them, i.e. if it contains guaranteed non-zero member variables or pointers to dynamic memory. However providing these functions explicitly reduces the risk that the programmer adds such a feature to a working class and that everything stops working unexpectedly.

If the programmer does not actually need one of these member functions, then it is perfectly acceptable to declare it as being private. The compiler will produce an error message if the function is called, even implicitly. For example, if the copy constructor is private the compiler cannot use it for parameters, which are passed " by value" (see later). If the function is never used then the programmer does not need to supply an implementation of the function and the declaration will prevent the compiler from supplying its own functions.

If the files for each class are based on a proforma file, or generated using an utility as mentioned in Rule 25, then skeleton declarations and definitions for these functions can be provided automatically. See appendix C for a suggested skeleton.

Rule 69. A class should always declare a constructor

The "default" constructor, i.e. the one with no arguments, is provided if no other constructors are explicitly declared. The "default" constructor supplied by the compiler will initialise data members to zero. For many classes this may be acceptable but there are cases where member variables may not take a zero value, as in the example below:

```
class CPolygon
{
    // …
        int m_numberOfSides;    // must be >= 3
};
```

The programmer should always declare a constructor and explicitly initialise the member variables appropriately.

Rule 70. A class should always declare its destructor.

The programmer should always declare a destructor in order to guarantee that memory is reclaimed correctly when an object is destroyed. Consider a class which represents a name.

```
class CName
{
    // …
```

```
        char *m_name;
};
```

The member variable points to dynamic memory. Assume that the member variable is initialised to NULL in the default constructor and to a particular name by providing a constructor with a string argument:

```
CName::CName()
{
    // default constructor
    m_name = NULL;
}

CName::CName(const char *nameString)
{
    assert(nameString != NULL);

    int length = strlen(nameString);

    m_name = new char[length + 1];
    strcpy(m_name, nameString);
}
```

As we have provided a means for allocating dynamic memory we must also provide a means for returning that memory to the system when it is no longer required and thus avoiding a memory "leak". This is obviously a task for the destructor:

```
CName::~CName()
{
    delete[] m_name;    // also handles m_name == NULL
}
```

Rule 71. If a class contains any virtual member function then also its destructor must be virtual.

Consider the following example:

```
#define LEN 80

class GraphicObject {
    public:
        virtual void display() = 0;    // Pure virtual
        GraphicObject() {
            // …
        }
        ~GraphicObject() {
```

```
            // …
        }
};

class Rectangle : public GraphicObject {
    public:
        Rectangle() {
            title = new char[LEN+1];
            strcpy(title, "(empty)");
            x1 = 0; y1 = 0;
            x2 = 0; y2 = 0;
        }
        ~Rectangle() {
            delete[] title;
        }
        void display() {
            // …
        }
    private:
        int x1, y1;
        int x2, y2;
        char * title;
};


// …

Rectangle * r1 = new Rectangle();
GraphicObject * r2 = new Rectangle();

// …

delete r1;          // OK:  ~Rectangle() is called
delete r2;          // Wrong: only ~GraphicObject is called
```

The problem, when deleting `r2` is that, because the `GraphicObject` destructor is not virtual, it is itself called rather than the `Rectangle` destructor.

When virtual member functions are used this is because the derived classes are most probably going to be accessed via a pointer to their common base class. In this situation they will be probably destroyed via this pointer to their base class; if the destructor of the base class is virtual, then the actual destructor of the derived class is called.

Rule 72. A class should always declare a copy constructor.

Consider the following non-member function, which takes a CName object as parameter:

```
Bool_t IsNameInDataBase(CName name)
{
    // …
}
```

This appears to be innocent enough but the compiler-supplied bitwise copy constructor give rise to a memory problem. When the "`name`" parameter is passed into the function in the example above it is passed "by value" which means that the compiler makes a copy of the actual object using the copy constructor. All manipulation within the function occurs on this local copy and is destroyed when the thread of execution returns from the function.

When the actual parameter is copied into the local variable both objects contain a pointer to the same area of memory. When the local variable is destroyed this area of memory os returned to the system leaving the actual parameter with a "dangling" pointer.

To avoid the problem with passing objects "by value" the programmer should explicitly define a copy constructor, which takes a copy of the dynamic memory part, as in the example below:

```
CName::CName(const CName & sourceName)
{
    int length = strlen(sourceName.m_name);

    m_name = new char[length + 1];
    strcpy(m_name, sourceName.m_name);
}
```

This problem does not occur when the parameter is passed "by reference", i.e. when the parameter is a pointer or reference argument.

Rule 73. The layout of ordinary function declarations should be used for member functions, with modifications for constructors with initialisation lists.

The layout of ordinary function declarations can be applied to member functions. A constructor which uses explicit initialisation of its base class or members needs an extension to this layout:

```
class CExample
{
    public:
        CExample(int x, int y);
    protected:
```

```
    private:
        int m_x;
        int m_y;
};

CExample::CExample(
    int x,
    int y
) : m_x(x), m_y(y)
{
    // body of the constructor, if any
}
```

Rule 74. Constructor functions which explicitly initialise any base class or member variable should not rely on a particular order of evaluation.

When using an initialisation list as in the previous example, the programmer must not initialise a data member in terms of another data member, which has been previously initialised using a parameter. If we extend the previous example to include a constructor which initialises both member variables to the same value:

```
CExample::CExample(
    int z
) : m_y(z), m_x(m_y)
{
    // body of the constructor, if any
}
```

This will not give the desired result `m_x` and `m_y` both being equal to `z` because the initialisation takes place in the order of declaration of the member variables. Therefore the `m_x(m_y)` initialisation is performed before the `m_y(z)` initialisation resulting in `m_x` being equal to zero and `m_y` equal to `z` instead of both being equal to `z`.

Rule 75. Objects should be constructed and initialised immediately if possible rather than be assigned after construction.

It is more efficient to create an object and initialise it immediately using a copy constructor than it is to construct and object and then assign to it later. This becomes more important as the class of the object becomes larger or more complex because the effort needed to construct one object only to tear it all down in order to assign the contents of another object may be significant.

```
CComplicatedClass newOne(oldOne);
```

```
// or
CComplicatedClass newOne = oldOne;

// are prefereable to
CComplicatedClass newOne;
newOne = oldOne;
```

12.3 THE ASSIGNMENT OPERATOR(S)

Rule 76. The class should always declare an assignment operator.

Remember that the compiler-supplied functions provide "bitwise" operations on the objects. Consider the following fragment of code:

```
CName name("Fred Smith");       // name.m_name points to "Fred
Smith"

void SomeFunction(void)
{                               // new scope
    Cname copy;                 // copy.m_name points to NULL;

    copy = name;                  // copy.m_name points to "Fred
Smith"
}                               // end of scope – copy destroyed
```

The variable "name" is constructed using the string provided and memory is allocated within the object for a copy of the string argument. The variable "copy" is constructed but it contains a pointer to NULL. In the absence of an explicit assignment operator when "name" is assigned to "copy" the compiler-supplied assigned operator is called. This makes a "bitwise" copy of "name" into "copy". This means that both "name" and "copy" contain a pointer to the same memory area (with value "Fred Smith"). At the end of its scope, the destructor is called for the "copy" variable. The memory which pointed by "copy" (and therefore also by "name") is returned to the system leaving "name" containing a "dangling" pointer, i.e. one which points to an undefined area of memory. What happens after this depends on the rest of the program, but it is unlikely to be what the programmer expects and could prove very difficult to debug.

To avoid this problem the programmer needs to explicitly declare an assignment operator function for the class. An initial version of this might look like the example below (although this will be defined in a later rule).

```
Cname & Cname::operator=(const CName & sourceName)
{
    // return the current memory to the system
    delete[] m_name;
```

```
        // allocate memory for the copy
        int length = strlen(sourceName.m_name);
        m_name = new char[length + 1];

        // copy from the source to this
        strcpy(m_name, sourceName.m_name);

        return(*this);
    }
```

Rule 77. The assignment operator(s) must check for assigning an object to itself.

The assignment operator given in the previous rule plugs one set of memory leaks compared to the compiler-supplied bitwise version when dealing with classes which contain pointers to dynamic memory. However it introduces a potential problem with data loss if the programmer assigns an object to itself because the dynamic memory pointed to by the target object is returned to the system before the memory of the source is copied. If both source and destination are the same this means that the information in the dynamic memory will be lost.

It is important that the assignment operator(s) check whether the source and the destination refer to the same object[1] before irrevocable changes are made to the data contained, as in the example below:

```
Cname & Cname::operator=(const CName & sourceName)
{
    // check whether assigning sourceName to itself
    if (this == &sourceName)
    {
        return (*this);
    }

    // return the current memory to the system
    delete[] m_name;

    // allocate memory for the copy
    int length = strlen(sourceName.m_name);
    m_name = new char[length + 1];

    // copy from the source to this
```

[1] Note that this test for object equality checks whether the source and destination objects reside at the same address in memory. For classes that are derived using multiple-inheritance there is no guarantee that pointers to the same object point to the same address in memory if pointers for different base classes are used.

```
    strcpy(m_name, sourceName.m_name);

    return(*this);
}
```

Rule 78. The assignment operator(s) must also assign base class member data.

Care should be taken that assignment operator(s) for a derived class must also handle the member data of the base class. This member data should not be copied on member by member basis because this means that any modifications to the base class must be explicitly handled in the derived classes. In any case, there may be private data members of the base class to which the derived class does not have access. To work around this problem the assignment operator of the derived class should make use of the assignment operator of the base class. Remember that the base class assignment operator can be a protected member function of the base class so that it is available to derived classes but not to the rest of the world. An example of this is provided here below:

```
class CDerived : public CBase
{
    // …
    CDerived    &    CDerived::operator=(const    CDerived    &
sourceDerived);
};


CDerived & CDerived::operator=(const CDerived & sourceDerived)
{
    // check whether assigning sourceDerived  to itself
    if (this == &sourceDerived)
    {
        return (*this);
    }

    // assign the data members of the CBase part of this
    // using the operator= function defined in CBase, i.e.
    // either CBase::operator(sourceDerived) ; or
    ((CBase) * this) = sourceDerived;

    // handle CDerived data memebers here
    // …

    return (*this);
}
```

Rule 79. The assignment operator(s) should return a reference to the object.

In the previous rules the assignment operator returned a reference to the object

but without any explanation. The reason is quite simple: it allows for a series of assignments in one statement:

```
sonsName = fathersName = grandFathersName;
```

Rule 80. Symmetric operators, with the exception of assignment operator, should be defined as friend functions. All asymmetric operators (i.e. (), [], unary * and unary ->) must be defined as member functions.

Operators can be defined either as member functions or as friend function. Because member functions are inherently asymmetric (i.e. they treat the Object in the argument list and `this` in different ways), it is recommended to use friend functions to implement symmetric operators and member function to implement the others.

12.4 GENERAL MEMBER FUNCTION GUIDELINES

Rule 81. Member functions, which do not alter the state of an object, shall be declared "const".

If a member function does not affect the internal state of an object, i.e. provides information about an object or calculates something derived from the contents of the object, then the member function should be declared as "const". This allows the compiler to issue error messages if the programmer subsequently tries to modify the objet from within the function.

```
int CName::Length() const
{
    int length = strlen (m_name);

    return m_name;
}
```

Rule 82. Public member functions must not return non-cost references or pointers to member variables of an object.

Rule 68 states that non-member variables may be public so that access to them can be controlled. The programmer must ensure that there are no ways of changing these member variables without using the intended interface.

If a member function returns a non-const reference or pointer to a member variable the programmer may inadvertently change the value of a member variable by using this reference or pointer. As there are other routines to limit the access to the member variables this is probably not what the class designer intended. Such indirect

access to member variables can result in corruption if other member variables are interdependent on the one being changed.

Rule 83. A function may not return a reference to memory, which it has allocated.

To avoid the problem of returning a pointer or reference to a local variable (see Rule 66) the programmer may be tempted to allocate dynamic memory within a function and return details of this memory to the caller. If the function returns a pointer to the memory the programmer is able to see that the caller must take care of returning this memory to the system.

However, if the function returns a reference to the memory it is easy for the programmer to forget that this memory must be returned to the system (especially if this function has been provided by a different programmer). Since the function returns a reference, the programmer is also able to use this reference as it where an object returned by value and it is easy to introduce memory leaks.

For example, consider the case where CName objects can be added together to form a new CName object (i.e. we are not concatenating one CName object onto another and therefore changing the original value of the object). If we consider part of the CName class definition and a non-member function which exhibits this problem:

```
class CName
{
    // …
        // call the next function append rather than operator+
        // to avoid confusion with the non-member function
        CName & append(const CName & sourceName);
    private:
        char *m_name;

        // declare the dubious non-member operator+ function as a
        // friend so that is can access the private data members
        friend CName & operator+(const CName & leftName,
                                 const CName & righName);
};

CName & CName::operator+(const CName & leftName,         // i.e.
leftname + RightName
                         const CName & rightName
)
{
    // first create copy of leftName in dynamic memory using
    // the CName copy constructor
    CName *bothNames = new CName(leftName);

    // add right name to copy using the add member function
    // which hides the direct string manipulation
```

```
    bothNames->append(rightName);

   // dereference the pointer and return it
   return (*bothNames)
}
```

Although this function would appear to do what is wanted, the caller must take care of returning the memory to the system. In the simple case of adding two names this may be fairly trivial:

```
CName firstName("John");
CName middleName("James");
CName lastName("Smith");

CNames & bothNames = firstName + lastName;
delete &bothName;
```

However, when the expression becomes a little more complicated the programmer does not have easy access to the returned objects because the values are not saved where the programmer can get at them:

```
CName & allNames = firstName + middleName + lastName;
// …
delete & allNames;     // this doesn't reclaim all memory
```

The compiler has broken this down into sub-expressions and has created a temporary variable to contain the result of one of these sub-expressions. It is the memory corresponding to this temporary, anonymous variable which has been lost. The programmer can break the expression down manually so that the results of each sub-expression are available:

```
CName & tempName = firstName + middleName;
CName & allNames = tempName + lastName;
// …
delete &tempName;
delete &allNames;
```

This approach relies on detailed knowledge on the part of the programmer concerning the memory handling within the function. The function cannot be used in the most intuitive way without leading to memory problems. Therefore this type of functions should be avoided and the programmer should be forced to work within the constraints of functions, which are "memory-safe".

```
CName allNames;                          // create allNames – this
will be
                                        // destroyed automatically
at the
                                        // end of scope
allNames.append(firstName);
allNames.append(middleName);
allNames.append(lastName);
```

Rule 84. Member functions shall only be declared as "inline" if the need for optimisation has been identified.

As mentioned in Rule 7 the programmer should keep the code simple and only optimise when a need for optimisation has been demonstrated. Inline member functions are a particularly problematic area. The compiler determines how "complex" the process of inlining the function will be depending on such criteria as the complexity of the function itself, whether it is a virtual function, and also on its interaction with other, possibly inline, functions.

The "`inline`" keyword is merely a hint to the compiler and the compiler is free to ignore it. In the event of this happening the compiler will convert the function into a static function in each module, which uses it. This will therefore result in an increase in size of the executable, without any of the benefits of optimisation.

If the compiler honours the request to inline the function it will substitute the code of the function directly at the calling site. This will produce some improvement in the underlying speed because the no actual function call takes place. For large functions, or ones which are called many times there will be a corresponding increase in the size of executables.

On systems where memory is limited, a significant increase in size may result in a drop of performance as the executable code is swapped in and out of memory.

Note that some debugging tools may not be able to handle inline functions.

Rule 85. A function shall not be declared as "inline" within the class declaration itself.

The class declaration in the interface file should expose to the outside world as little as possible of the internal details of that class. Therefore the implementation of any inline member function should not appear inside the class declaration itself. If possible the definitions of inline member functions should appear after the class declaration, or in a separate implementation file which is the #included in the interface file.

The actual mechanics of this process are not specified here because of the

possible constraints made by the development environment. One suggested method is given in appendix D.

CHAPTER 12
CLASSES [C++ ONLY]

This page is intentionally left blank

CHAPTER 13
TEMPLATES [C++ ONLY]

Template classes appear to be the answer to many programmers' prayers but in reality are not as straightforward as they might at first appear, especially when trying to provide a truly general template class.

The design of a template needs some forethought. The designer must determine whether the parameter for the template is a pointer to an object, a reference to an object or the object itself. This can make a big difference to the implementation and efficiency of the template such as when copying objects which are passed "by value" (see Rule 74) which also requires the class to have a public copy constructor. Any class, which is used as a template parameter, must also satisfy all of the requirements of the template. For example a general purpose template may declare a function which uses the `operator<<` member function of the parameter class which means that the parameter class must have declared an `operator<<` member function itself even if this feature of the template class is not actually used.

Rule 86. Templates should be encouraged as a convenient way of reusing code.

(Given the templates were properly designed, as discussed in the previous paragraph).

Rule 87. If templates are used then Auto_ptr pointers should be preferred to normal pointers.

Recommendations and examples to be taken from "The C++ Programming Language", pages 367-369 (Appendix A, #12).

CHAPTER 13
TEMPLATES [C++ ONLY]

This page is intentionally left blank

CHAPTER 14
EXCEPTIONS [C++ ONLY]

Rule 88. **A function which can issue exceptions (i.e. has a "throw") clause in its declaration) can only be called:**

1. **from within a "try-catch" construct catching all those exceptions;**

2. **from within a "try-catch" construct catching some of those exceptions where the other are declared in the "throw" clause of the function containing the "try-catch" construct;**

3. **from within another function which exports (via the "throw" clause int its declaration) exactly the same exceptions.**

This rule forces to make very well clear in the code (and in the design) how exceptions flow from a portion of code to another. Example.

```
class ExBase {
private:
  char * name;
public:
  ExBase (char * nm) { name = nm; }
  char *name() { return name; }
  void announce() {
    cout « name() « "!" « endl;
};

class Ex1553 : public ExBase {
public:
  Ex1553() : ExBase("1553 exception") {
    announce();
  }
};

class ExRs422 : public ExBase {
public:
  ExRs422 : ExBase("RS422 exception");
    announce();
  }
};

// Incorrect definition of Foo
void Foo() {
  // …
```

```
  throw Ex1553();
  // …
  throw ExRs422();
  // …
}

// Correct definition of Foo
void Foo() throw (Ex1553, ExRs422) {
  // …
  throw Ex1553();
  // …
  throw ExRs422();
  // …
}

// Incorrect use of Foo (Foo is called without controlling the
exceptions
// it may throw
void Goo() {
  // …
  Foo();
  // …
}

// Two correct ways of calling Foo

// All exceptions generated in Foo are caught in Goo
void Goo() {
  // …
  try {
    Foo();
  } catch (Ex1553) {
    cout « "caught Ex1553" « endl;
  } catch (ExRs422) {
    cout « "caught ExRs422" « endl;
  } catch (ExBase eb) {
    cout « "caught " « eb.name() « endl;
  }
  // …
}


// Goo cannot catch ExRs422 (it does not know what to do with it)
void Goo() throw (ExRs422) {
  // …
try {
    Foo();
  } catch (Ex1553) {
    cout « "caught Ex1553" « endl;
}
         // …
}
```

Rule 89. **Every C++ program using exceptions must use the function set_unexpected() to specify which user defined function must be called in case a function throws an exception not listed in its exception specification.**

Rule 90. **Every C++ program using exceptions must use the function set_terminate() to specify which user defined function must be called if an handler for an exception cannot be found.**

CHAPTER 14
EXCEPTIONS [C++ ONLY]

This page is intentionally left blank

CHAPTER 15
CASTING [C++ ONLY]

Rule 91. The programmer should use the new C++ cast operators rather than the traditional C cast.

The traditional C method of casting simple variables and pointers from one type to another is still available in C++ but its use should be discouraged. The traditional C cast is an instruction to the compiler to override any type information, which it may hold about an expression.

Note that this can lead to portability problems because a cast, which may be valid on one system, may not be valid on another but the compiler assumes that the programmer knows best.

The C++ standard defines four new cast operators which are more specific in their operation and therefore the intention of the cast is clearer:

- `static_cast<type>(expression)`
- `const_cast<type>(expression)`
- `dynamic_cast<type>(expression)`
- `reinterpret_cast<type>(expression)`

The first is equivalent to a traditional C cast but is more obvious in the code.

The second allows the programmer to remove the `const` or `volatile` nature of the expression.

The third provides a means of testing whether a class instance object belongs to a particular class within the class hierarchy.

The fourth is intended for particularly complicated and non-portable casting such as that one concerning pointers to functions.

CHAPTER 15
CASTING [C++ ONLY]

This page is intentionally left blank

CHAPTER 16
STANDARD TEMPLATE LIBRARY [C++ ONLY]

The Standard Template Library (STL) is not part of the C++ standard but has been developed in parallel with it. It provides a series of templates, which the programmer can use for implementing common constructs such as, sets, lists, the iterators which work with them, etc.

CHAPTER 16
STANDARD TEMPLATE LIBRARY [C++ ONLY]

This page is intentionally left blank

CHAPTER 17
NAMESPACES [C++ ONLY]

No special rules are given here.

CHAPTER 17
NAMESPACES [C++ ONLY]

This page is intentionally left blank

CHAPTER 18
EXPRESSIONS

18.1 CONDITIONAL EXPRESSIONS

Rule 92. Conditional expressions must always compare against an explicit value.

The traditional C idiom has always favoured brevity in code and this is extended to the implicit comparison of values against zero, which is also used to denote false. In some cases this can be counter-intuitive and the programmer reading the code must expend some effort to understand what is actually happening.

The other area where this can be problematic is when dealing with the return codes from functions. In many cases there is a range of return codes which may indicate different levels of success or failure of the function and not just zero (false) and non-zero (true).

Different routines are not consistent in their use of return codes to indicate success and failure. Many routines which deal with pointers return NULL (i.e. zero) to indicate failure, while many others use a negative number to indicate failure. The common string comparison routine `strcmp()` uses zero to indicate equality!

Routines, which use enumeration as the return value, can suffer when a new member is added to the enum, which may cause the renumbering of the existing values, and invalidate any implicit comparisons.

The programmer is therefore encouraged to use explicit comparisons in the code so that the meaning is clear, and to safeguard against changes in the underlying implementation of functions and their return types.

```
if (Function(parameter) == SUCCESSFUL)
```

is guaranteed to continue working as expected even if the actual value associated with the return code `SUCCESSFUL` is ever changed. It doesn't matter whether `Function()` returns a negative number, zero, or a positive number as long as `SUCCESSFUL` has the appropriate value. The symbolic name also helps to clarify the code. On the other hand, the following fragment is unclear and will not continue to work correctly if the return value is changed to or from zero.

```
if (Function(parameter))
```

Note also that the language standards define zero as meaning false. They do not define a particular non-zero number to mean true. It is theoretically possible to fall foul of a compiler if a symbolic constant for true is defined to be a particular value. For example if `MyFalse` is declared to be 0 and `MyTrue` to be 1 and if value is 2 then both `value!=MyFalse` and `value!=MyTrue` are true at the same time. This does not lead to intuitive code!

18.2 ORDER OF EVALUATION

Rule 93. The programmer shall make sure that the order of evaluation of the expression is defined by typing in the appropriate syntax.

The order of evaluation of individual expressions within a statement is undefined, except when using a limited number of operators.

The comma operator in "`expression1, expression2`" guarantees that expression1 will be evaluated before expression2. Note that the comma operator is not the same as the comma used as a separator between function arguments.

The logical-AND operator in "`expression1 && expression2`" guarantees that expression1 will be evaluated first. expression2 will only be evaluated if expression1 is true.

The logical-OR operator in "`expression1 || expression2`" guarantees that expression1 will be evaluated first. expression2 will only be evaluated if expression1 is false.

The conditional operator in "`expression1 ? expression2 : expression3`" will evaluate expression1. If it is true expression2 is evaluated, otherwise expression3 is evaluated.

All expressions, which appear as function arguments, are evaluated before the function call actually takes place. Note that the order of evaluation of the arguments is not specified.

In addition, a "full expression" is the enclosing expression that is not a sub-expression. The compiler will evaluate each full expression before going further.

The operators, etc. which are explained above are known as "sequence points" and can be used to determine the order of evaluation of the expressions in some

statements. However the order of evaluation of expressions between each sequence point is undetermined.

This means that different compilers may produce different answers for particular statements. This is most notable when using expressions which contain side-effects. For example the outcome of the following two statements is undefined.

```
firstArray[i] = secondArray[i++];
```

Rule 94. [C++ only] The programmer must not override the comma, &&, || and ?: operators.

In C++ the programmer has the opportunity of overriding the comma, logical-AND, logical-OR and the conditional operators. This is not recommended. Not only would it be hard for a programmer to overload these operators to have a similar intuitive meaning, but once overloaded these operators lose their function as sequence points. This means that the code using these new operators is unlikely to work in the same way as the rest of the code.

18.3 USE OF PARENTHESES

Rule 95. The programmer must use parentheses to make intentions clear.

As well as considering the order of evaluation of a statement, the programmer must also consider the precedence of the operators used in the statement because these affect the way the compiler breaks down the statement into expressions. For example, the relative precedence of the addition and multiplication operators mean that the expression

```
a + b * c
```

is really treated as

```
a + ( b * c )
```

rather than

```
( a + b ) * c
```

The programmer is advised to make explicit use of parentheses to reduce any possible confusion about what may have been intended and what the compiler will assume by applying its precedence rules.

Rule 96. The programmer must always use parentheses around bitwise operators.

The use of the bitwise operators is not always intuitive because they appear to offer two different types of behaviour. On the one hand they behave like arithmetic operators while on the other hand they behave like logical operators. The precedence rules may mean that a bitwise operator behaves as expected when used in one context, but not when used in another context. The safest course is to use parentheses with the bitwise operators so that the code is clear.

```
if (statusWord & PARTICULAR_STATUS_BIT)
```

lacks an explicit comparison (see Rule 92) so the programmer modifies it to what would initially appear to be an equivalent form:

```
if (statusWord & PARTICULAR_STATUS_BIT != 0)
```

Unfortunately the programmer has just introduced an error into the code! The precedence rules mean that the bitwise-AND operator has a lower precedence than the `!=` comparison operator and as a result `PARTICULAR_STATUS_BIT` is tested against zero and the outcome is then combined with the `statusWord` using the bitwise-AND. The programmer must use parentheses to restore the code to working order:

```
if ((statusWord & PARTICULAR_STATUS_BIT) != 0)
```

18.4 USE OF WHITE SPACE

Rule 97. Do not use spaces around the "." and "->" operators or between a unary operators and their operands.

These operators are tightly coupled with their operands and the precedence rules mean that they are evaluated before the other operators. Adding spaces between the operator and the operand makes them appear to be more loosely coupled than they really are and this can lead to confusion.

Rule 98. Other operators should be surrounded by white space.

In contrast with the previous rule, other operators should be surrounded by white space in order to give additional visual separation of the operands in order to show that they are more loosely coupled. Note that parentheses may be used to group operands for evaluation purposes and white space may be omitted for "tightly-coupled" operands.

```
y = (a * x * x) + (b * x) + c;  // or y = a*x*x + b*x + c;
```

However, when in doubt about operator precedence, use parentheses!

CHAPTER 18
EXPRESSIONS

This page is intentionally left blank

CHAPTER 19
MEMORY ALLOCATION

The C memory allocation routines `malloc()`, `calloc()` and `realloc()` return a pointer to dynamic memory if they succeed. This pointer also provides the system with the means of accessing the size of the memory allocated, etc. although this information is not readily available to the programmer. The system has no way of forecasting what the programmer will do with the pointer or memory but it assumes that the programmer will use the pointer and the memory wisely and safely, and will return it to the system after use.

If the programmer does not use the pointer and memory wisely, the system has no way of recovering that memory. If the value contained in the pointer variable is lost, i.e. if it is overwritten or the variable goes out of scope, the system will no longer have access to its housekeeping information about the memory and therefore the memory can never be reclaimed. An area of memory, which cannot be reclaimed, is known as a memory leak.

If the programmer inadvertently writes outside the area of memory which was requested it is possible to corrupt other areas of user data, or even the housekeeping information about the memory which the system maintains. This is likely to result in unexpected behaviour by the program possibly in a completely unrelated area of code. Tracing the error back to its source can prove difficult.

Rule 99. The return values of memory handling routines must always be checked.

If a memory allocation routine fails then the pointer which is returned will be invalid and using it to access memory is likely to result in a catastrophic error. Therefore the return values should always be checked[2].

Rule 100. Dynamic memory allocated using malloc() should be returned to the system using free().

Even in environments with virtual memory the total amount of memory available to the programmer is still a finite resource. Therefore memory which has been allocated using `malloc()`, `calloc()` or `realloc()` and which is no longer needed should

---

[2] Note that older version of the draft C++ standard allow new to return a NULL pointer in the event of a failure. More recent versions specify that new should throw an exception if it fails.

be returned to the system using `free()`.

Rule 101. The programmer must ensure that a pointer to dynamic memory is not lost when using realloc().

The `realloc()` function is used to request a new area of dynamic memory. It takes a pointer to existing memory and the size of new memory required and returns a pointer to the new memory. However the standard C idiom for using `realloc()` can result in a catastrophic memory leak if the memory allocation fails:

```
pMemory = realloc(pMemory, newSize);
```

The reason that this is a dangerous practice is that `realloc()` returns NULL if it is unable to honour the request for new memory. In the example above the pointer to existing memory will be overwritten with NULL if `realloc()` fails. Even if the following code tests the new pointer value to see whether the request was successful, the pointer to the existing memory is lost, along with the data contained within it. This gives rise to a memory leak, but more importantly the programmer has lost any access to the data held in the memory.

To ensure that the pointer to the data is still available even in the event that `realloc()` fails the programmer is advised to use:

```
pNewMemory = realloc(pOldMemory, newSize);
```

The programmer therefore still has access to the old data in the memory and can take corrective action to avoid further data loss.

Rule 102. [C++ only] Memory allocation using new/delete should be used instead of malloc()/free().

The simple reason for this rule is that `new` and `delete` are part of the C++ environment and know about constructors and destructors. Objects, which are allocated using new, can be constructed and initialised automatically. Similarly the destructor for the object is called when `delete` is used.

`malloc()` and `free()` are C library routines which only handle memory allocation and know nothing about the construction and destruction of objects in that memory.

In the event that the programmer must use C functions from within the C++ code, such as during the porting of an application from C to C++ or when using third party

library routines, care should be taken to ensure that the memory allocated using `malloc()` et al is returned to the system using `free()`. Memory allocated using new should be returned to the system using `delete`.

The C and C++ memory handling schemes should not be intermixed when allocating and de-allocating individual areas of memory.

Rule 103. [C++ only] If the call to new uses [] then the corresponding call to delete must also use [].

If an object is allocated using `new` the pointer to the object provides the system with the indirect means of discovering information about the memory needed for that object. This information is used by `delete` when returning the memory to the system.

A similar process is used for an array of objects. If an array of objects is allocated the programmer must use `new` with `[ ]` in order to tell the system about the size of an individual object and also information about the array of objects. The ordinary `delete` function (i.e. without `[ ]`) only knows how to handle a single object and not an array of objects. The delete function using `[ ]` does know how to handle an array of objects and is able to return the memory for the whole array back to the system.

To avoid problems with memory leaks or corruption, matching pairs of new/delete or `new[ ]/delete[ ]` should be used when allocating and de-allocating each area of memory.

Rule 104. Every C++ program using dynamic memory allocation must use the function set_new_handler() to specify which user defined function must be called in case of memory allocation failure.

CHAPTER 19
MEMORY ALLOCATION

This page is intentionally left blank

CHAPTER 20
ERROR HANDLING

There are several classes of error for which the programmer may be able to take some sort of corrective action. What action is taken may depend on whether the program is being run under a development environment or whether it is used under "real world" conditions.

There are some errors, which will cause the abnormal termination of the program such as the corruption of some internal data structures etc. due to programmer error. Apart from trying to avoid making the error in the first place there is very little that the programmer can do to recover from such an error. However the programmer may be able to detect the causes of the error and issue a diagnostic message before the program terminates, especially during development when additional diagnostic code may be available in the program.

Other error conditions may be detectable and not immediately fatal to the program. The user may be able to intervene to alleviate the error before the program continues any further. Examples of this would be being unable to write data to disk because the disk is currently full, or being unable to open another document because insufficient memory is available. This type of error can be handled by allowing the user to make some manual corrective action, such as changing disk or deleting files, or by allowing the user to save an existing document and therefore freeing memory before opening the next one.

During development the programmer should make use of additional diagnostic code to detect potential errors as soon as possible. These may simply be due to the incomplete nature of the code as it is being developed. On the other hand these may be due to errors in the program and their origin should be established as quickly as possible. It is far better to detect errors during development and correct them than it is to release the software to the customer and for the error to be discovered while the system is "live".

Rule 105. The programmer should validate function parameters where possible.

Many errors can be detected early if during development the programmer includes the assert() macro or other diagnostic code in each function which validates the assumptions made about the parameters. If a parameter is invalid, e.g. a pointer which is NULL or a value which is out-of-range, it can indicate that problems exist in the calling code.

Rule 106. The return values of library functions should be checked for errors.

The programmer must never assume that all library routines will always succeed. Some routines may set the return value to an error code, and others may set a separate error variable. The programmer will need to check the appropriate variable. The first case usually involves "out of band" data which will cause problems elsewhere in the code if it is undetected. In the second case the return values may be unmodified by the routine if there is an error but may still contain the remnants of an earlier successful invocation which may be no longer applicable. If the error code is not checked and appropriate action taken the outcome may be unexpected.

Note that there are some routines, such as `atoi()`, which will happily accept "invalid" input and still produce a "valid" result.

Rule 107. Diagnostic code should be added to all areas of code which "should never be executed".

Many programmers still include code for exceptional cases, which should never be executed during normal use, such as default clauses in switch statements (see Rule 23). Simply adding a comment saying that such code should never be reached does not help to detect whether that code has actually been executed. The programmer should add diagnostic code in order to detect whether such an error occurs.

CHAPTER 21
THE USE OF THE PREPROCESSOR

Rule 108. The programmer should use #include system header files using <name.h> and user header files using "name.h"

There are two different reasons for using different #include semantics for system header files and user header files. The first is that the programmer can see from the use of angle brackets or double quotes whether the header file is a system header file or a user header file as this might not be obvious from the name alone.

The second reason is that on some systems the different #include semantics imply slightly different behaviour when it comes to searching for the include file in the file system.

Rule 109. The #include line may not contain the full path to the header file.

If the programmer specifies the full path to a header file then the source file will need to be modified when the header file is moved. Different operating systems also use different ways of specifying a full path and this will therefore need to be changed when porting the software from one system to another.

Rule 110. Use conditional compilation to select diagnostic or non-portable code.

As mentioned in Rule 9, diagnostic code should be enclosed within #ifdef/#endif pairs so that it can be removed from the release version of the program while leaving it available for development purposes.

Non-portable or system-specific code should also be enclosed within suitable #ifdef/#else/#endif pairs. This allows different versions of code to be compiled without the overhead of a separate file for each system, and the configuration problem, which this may cause. Such conditional compilation can allow for an alternative version of the code, or for a series of different versions.

```
#ifdef INTEGER_ARITHMETIC
    /*
     *  Faster,  but  less  accurate  version  using  integer
arithmetic.
     */
#else
    /*
     * Standard floating point version.
     */
```

```
#endif /*INTEGER_ARITHMETIC*/

#ifdef UNIX
    /* Unix specific code */
#elif DOS
    /* DOS specific code */
#else
    ERROR: This code only works on Unix and DOS
#endif
```

Rule 111. Pre-processor macros, which contain parameters or expressions, must be enclosed in parentheses.

The constants and macros provided by the pre-processor are not really part of the language and are simple substituted directly into the source code. Therefore care should be taken when using expressions (even in constants!). The body of the macro should be enclosed in parentheses.

```
#define TWO 1+1          /* should be (1+1) */
six = 3 * TWO;           /* six becomes 3*1+1 i.e. 4 */
```

If a macro contains any parameters, each instance of a parameter in both the macro declaration and in the macro body should be enclosed in parentheses. The parentheses around the parameter in the declaration protect the macro in the event that is called with an expression which uses the comma operator which would cause confusion about the number of parameters. The parentheses around the parameters in the macro body protect the macro from conflicts of precedence if the parameter is an expression.

```
#define RECIPROCAL(x) (1/x)     /* should be (1/(x)) */
half = RECIPROCAL(1+1);         /* half becomes 1/1+1 i.e. 2 */
```

Rule 112. Macros should not be used with expressions containing side effects.

This is another facet of Rule 93. The macro hides the underlying expression. The programmer may only see one expression (with possible side effect) when calling the macro, but underneath the macro body may use this expression several times. The order of evaluation of arguments before calling a function does not apply to a macro because it is not a function but merely a pre-processor convenience.

```
#define SQUARE(x) ((x)*(x))  /* nothing unusual */
z = SQUARE(y++);             /* z becomes y++*y++ i.e. what? */
```

Rule 113. The pre-processor may not be used to redefine the language.

CHAPTER 21
THE USE OF THE PREPROCESSOR

Programmers who are more comfortable using another programming language may be tempted to provide pre-processor macros, which map C and C++ constructs into some semblance of this other language.

```
#define IF    "if"
#define THEN  "{"
#define ELSE  "} else {"
#define FI    "}"

IF (x == 1)
THEN
    statement(s);
ELSE
    statement(s);
FI
```

This is expressly forbidden because the programmer is supposed to be using C or C++ and not some other language with which a future maintenance programmer may not be familiar. Certain constructs may not map directly from C into the other language and may therefore have restrictions on their use. Various support tools, such as syntax-aware editors, will be unable to work with the macros.

CHAPTER 21
THE USE OF THE PREPROCESSOR

This page is intentionally left blank

CHAPTER 22
PORTABILITY

22.1 DATA ABSTRACTION

Rule 114. The programmer should use "problem domain" types rather than implementation types.

The programmer is encouraged to add a level of data abstraction to the code so that variables are expressed in terms which relate directly to the problem domain rather than to the underlying "computer science" or hardware implementation.

The use of "`typedef`" can also improve maintainability because only one place in the code needs to be changed if the range of values for that type must be changed. For example the range of values may be changed by converting a variable, or set of variables, from "`short`" to "`unsigned short`".

Portability is improved because only one place in the code needs to be changed to make use of different hardware characteristics on different platforms.

```
typedef unsigned short AgeInYears_t;
typedef float KmPerHour_t;
```

22.2 SIZES

Rule 115. The programmer may only assume range(char) < range(short) <= range(int) < range(long).

ISO/IEC C standard defines a standard header file, limits.h, in which standard macro names are defined to denote the minimum and maximum values of the different integer types on that system.

The standard also provides the minimum range of values for a particular integer type, which is guaranteed to be portable between different implementations.

Note that the exact range allowed for a variable of type char depends on whether the implementation uses a signed char or an unsigned char implementation.

Rule 116. The programmer may only assume that range(float) <= range(double) <= range(long

double)

ISO/IEC C Standard also defines a standard header file, float.h, in which standard macro names are defined to denote the minimum and maximum values of the different floating point types on that system.

As for integer types, the default ANSI values provide the ranges which are guaranteed to be portable between implementations.

22.3 REPRESENTATION

Rule 117. The programmer may not assume knowledge of the representation of data types in memory.

Different systems may make use of different representations of data types in memory. Such differences may include differences in word ordering, byte ordering within a word and even the ordering of bits within a byte. The programmer should therefore avoid any specific knowledge of an underlying representation when manipulating the data because what may be valid on one system may not hold true on another system.

Rule 118. The programmer may not assume that different data types have equivalent representations in memory.

This rule follows from the previous section and the rule above. If the different types specify different ranges of values it is likely that they are represented differently in memory. The programmer may not assume that different types share a common representation in memory.

22.4 POINTERS

Rule 119. The programmer may not assume knowledge of how different data types are aligned in memory.

Different hardware platforms impose different restrictions on the alignment of data types within memory.

Rule 120. The programmer may not assume that pointers to different data types are equivalent.

Some hardware platforms actually use different pointer representations for different data types. Therefore assigning one pointer value for one type to a pointer variable for another type is not always guaranteed.

The only exception is the equivalence of void* pointers to pointers of other types.

Rule 121. The programmer may not mix pointer and integer arithmetic.

Pointers are not integers. The programmer must not treat pointers as integers. The programmer is prohibited from assigning integers (or integer values) to pointers and vice versa. If we assume the following declarations:

```
int intValue;
Thing_t thingArray[10];
Thing_t *thingPointer = thingArray;  // i.e. &thingArray[0];

intValue = thingPointer;     // DO NOT DO THIS IN REAL CODE!

intValue += 1;               // increase integer value by 1
thingPointer += 1;           // adjust pointer value to point to
                             // next thing, i.e. &thingArray[1]
```

At this point it is unlikely that intValue and thingPointer contain the same value because intValue has been incremented by 1 whereas thingPointer has been adjusted by the size of a Thing_t object (including possible adjustments for the alignment of Thing_t objects in memory).

22.5 UNDERFLOW AND OVERFLOW

Rule 122. The programmer must use a wider type or unsigned values when testing for underflow or overflow.

This addresses the case when the programmer is dealing with two integer variables, with a relatively narrow possible range (such as chars and shorts), and is worried about overflow or underflow. Then it is possible to convert the values into variables with a wider range in order to detect whether overflow or underflow occurs during the calculation.

```
if ( (long)shortValue + (long)shortValue > SHRT_MAX)  // overflow
```

This is not possible if the variables in question are of type `long` or have the same range as a `long`. See also Rule 115.

However it is possible for the programmer to code around potential problems of overflow or underflow when dealing with two unsigned integer values because this can be detected without relying on the underlying hardware implementation.

```
if (a+b < a || a+b < b)     // overflow has occured!
```

When handling one signed and one unsigned integer value the programmer can still detect potential problems by converting the signed value into an unsigned value and using unsigned arithmetic.

However, when dealing with two signed integer values what happens when overflow or underflow occurs is implementation dependent. The program may abort, it may set some error code, which can be tested, or it may simply continue.

If overflow and underflow are likely to be a problem the programmer may need to convert to using unsigned arithmetic to be able to guarantee detection.

22.6 CONVERSION

Rule 123. The programmer must be careful when assigning "long" data values to "short" ones.

As mentioned in Rule 115 and Rule 116 the ranges of the different integer types and floating point types can be different. Therefore the programmer must ensure that a value does not lie outside the permitted range of a particular type before assigning the value to a variable of that type.

CHAPTER 23
EMBEDDED C++

Rule 124. In case of selection of C++ for space on-board software, the subset "Embedded C++" shall be used.

This chapter intends to point to the "Embedded C++ Language Specification". One of the goals of this language, as taken out from their web page (Appendix A, #9) is to:

"Avoid those features and specifications that do not fulfil the requirements of embedded system design. The three major requirements of embedded system designs are:

- Avoiding excessive memory consumption
- Taking care not to produce unpredictable responses
- Making code ROMable"

"Embedded C++" is a pure subset of ISO/IEC14882 C++ standards and the following are the major restrictions put on normal C++:

- no exceptions
- no RTTI
- no namespaces
- no templates
- no multiple inheritance
- no virtual inheritance
- no dynamic memory allocation

This language is appropriate for the production of on-board software, especially if the criticality level is high.

CHAPTER 23
EMBEDDED C++

This page is intentionally left blank.

APPENDIX A BIBLIOGRAPHY

1) Space Engineering - Software Standard, ECSS-E-40A, 13-April-1999 (http://www.estec.esa.nl/ecss)

2) Guide to the software detailed design and production phase (ESA PSS-05-05), ESA, 1992

3) Kernighan and Ritchie, "C Programming Language (Second Edition)", Prentice Hall, 1988, ISBN 0-13-110362-8

4) Koenig, "C Traps and Pitfalls", Addison-Wesley, 1988, ISBN 0-201-17928-8

5) Harbison and Steele, "C, A Reference Manual (Fourth Edition)", Prentice Hall, 1995, ISBN 0-13-326232-4

6) C Standard, ISO/IEC 9899

7) Plum, "C Programming Guidelines", Plum Hall Inc., 1984

8) C++ Standard, ISO/IEC 14882

9) "Embedded C++ Specification", http://www.caravan.net/ec2plus/language.html

10) Plum and Saks, "C++ Programming Guidelines", Plum Hall Inc., 1991, ISBN 0-911-537-10-4

11) Eckel, "Thinking in C++", Prentice Hall, 1995, ISBN 0-13-917709-04

12) Stroustrup, Bjarne. The C++ Programming Language. 3e Edition. Addison-Wesley, 1997. ISBN 0-201-88954-4.

A high quality reference book including all the details about C++ use. Each chapter concludes with an advice section.

13) Stroustrup, Bjarne & Ellis The Design and Evolution of C++ Addison-Wesley, 1994. ISBN 0-201-54330-3.

This book from C++ father introduces the evolution of the language, and clarifies why some features have been added. The practical content will benefit to C++ programmers.

14) Stroustrup, Bjarne & Ellis The Annotated C++ Reference Manual. Addison-Wesley, 1990. ISBN 0-201-51459-1.

A complete and sophisticated reference book, although not young.

15) McConnell, Steve. Code Complete. A practical handbook of Software construction, Microsoft Press, 1993.

A huge manual introducing the basic of software building: standards, advises. A must.

16) Maguire, Steve. Writing Solid Code. Microsoft's Techniques for developing Bug-Free C Programs. Microsoft Press, 1993. ISBN 1-55615-551-4.

A book to support education intended to improve writing and debugging code. Can be read as a novel.

17) Meyers, Scott. Effective C++. 50 Specific Ways to Improve Your Programs and Designs. Addison-Wesley, Second Edition. 1998. ISBN 0-201-92488-9.

To complement education. 50 items are introduced in order to better understand advanced features of C++ design and coding.

18) Meyers, Scott. More Effective C++. 35 New Ways to Improve Your Programs and Designs. Addison-Wesley, 1996. ISBN 0-201-63371-X.

35 more items.

19) Cargil, Tom. C++ Programming Style. Addison-Wesley, 1992. ISBN 0-201-56365-7.

To go on with C++ learning and improve specific programming features.

20) Carroll, M. D./Ellis M.A. Designing and Coding Reusable C++. Addison-Wesley, 1995. ISBN 0-201-51284-X.

To refine the development from the reuse standpoint. Many ideas to manage component reuse.

21) Coplien James O. Advanced C++ Programming Styles and Idioms, Addison Wesley 1992, ISBN 0-210-54855-0.

A good reference book, often referenced, although old.

22) H.M. Deitel / P.J. Deitel. C++ How to program. Prentice Hall. Second Edition, 1998. ISBN 0-13-528910-6.

A learning book covering most of the C++ features. A pragmatic approach focussing on C++ features explanation. Numerous exercises.

23) Henricson/Nyquist, Industrial Strength C++ - Rules and Recommendations, Prentice

Hall, 1997, ISBN 0-13-120965-5.

A reference book on programming standards.

24) Horstmann, Cay S., Mastering Object-Oriented Design in C++, John Wiley & Sons Inc., 1995. ISBN 0-471-59484-9.

A learning book covering all the centre of interest of object oriented programming. Comes with a library.

25) Lakos, John, Large Scale C++ Software Design, Addison-Wesley, 1996, ISBN 0-201-63362-0.

To go on with learning large scale developments. It highlights the origin and the solution of many problems linked to interdependency between the components of an industrial sized system.

26) Lippman, Stanley B. et Josée Lajoie. C++ Primer, Third Edition, Addison-Wesley, 1998. ISBN 0-201-82470-1.

The last version brings the book to a reference, largely used for C++ teaching.

27) Steve Maguire. DEBUGGING THE DEVELOPMENT PROCESS Practical Strategies For Staying Focused, Hitting Ship Dates, And Building Solid TeamsMicrosoft; ©1994

CHAPTER 23
EMBEDDED C++

This page was intentionally left blank

APPENDIX B THE STANDARD COMMENT HEADER BLOCK

B.1 Suggested List of Keywords

The following sets of keywords are suggestions for use in the standard comment header blocks of interface and implementation files. Each project is allowed to define additional keywords as necessary.

Keywords for the interface ("*.h") files

Title	
SccsId	
Origin	
Author	
Reviewed	
Purpose	
Note	(Optional)
Changes	(Optional)

Keywords for the implementation ("*.c") files

Title	
SccsId	
Method	
References	(Optional)
Author	
Reviewed	
Note	(Optional)
Changes	(Optional)

The SccsId keyword could be replaced by a more suitable keyword if another source code configuration system is used.

If no source code configuration system is used, the optional keyword Changes shall be used to include information about when and why the file has been updated.

B.2 Interface (".h") file comment block

```
/*
 * Title:
 *    useful module
 *
 * SccsId:
 *    %W% %E% %U%
 *
 * Origin:
 *    ESADEMO Project, created by XYZ Company,
 *    contract number 4567/89/NL/DK
 *
 * Purpose:
 *    This module provides something that is useful to someone.
 *
 * Author:
 *    Arne Lundberg
 *
 * Reviewed:
 *    Kurt Olsson, 26 May 89
 *
 * NOTE:
 *    This module allocates memory which is never deallocated!
 */
```

B.3 Implementation (".c") file comment block

```
/*
 * Title:
 *    useful module
 *
 * SccsId:
 *    %W% %E% %U%
 *
 * Author:
 *    Arne Lundberg
 *
 * Reviewed:
 *    Kurt Olsson, 26 May 89
 *
 * Method:
 *        This module uses a finite state machine which is
implemented
 *    using a large switch statement.
 *
 * References:
 *     This algorithm is taken from "Some Book in the Library",
p25
 */
```

APPENDIX C THE PROFORMA CLASS DECLARATION

C.1 The Class Declaration (".h") file

```
//
// example.h
// interface file comment block omitted
//

#ifndef example_h
#define example_h 1

class CExample
{
    //  The  "standard"  functions  are  declared  as  private  by
default
    // and those which are needed will need to be promoted into
    // the appropriate access category. Note that skeleton
    // versions are provided in the implementation file.

    public:

    protected:

    private:
        CExample();                                    //  "default"
constructor
        CExample(const CExample& source); // copy constructor

         ~CExample();                          // destructor - may
need
                                             // to be made virtual

                                             // assignment operator
        CExample& operator=(const CExample& source);
};

#endif //example_h
```

C.2 The Class Implementation (".c") file

```
//
// example.c
// implementation file comment block omitted
//

#include "example.h"

//
// "default" constructor
//
```

```
CExample::CExample()
{
    // TODO: Initialise any member variables here
}

//
// copy constructor
//

CExample::CExample(
    const CExample& source   // object from which to copy
)
{
    // TODO: copy data members from source
}

//
// destructor
//

CExample::~CExample()
{
    // TODO: tidy up data members
}

//
// assignment operator
//

CExample& CExample::operator=(
    const CExample& source      // object from which to assign
)
{
    // check whether assigning source to itself
    if (this == &source)
    {
        return(*this);
    }

    // if this is a derived class then assign the data members of
    // the base class using the operator= member function of the
    // base class, i.e.
    // CBase::operator=(source);
    // or
    // ((CBase&) *this) = source;

    // TODO: tidy up data members and then copy from source

    return(*this);
}
```

CHAPTER 23
EMBEDDED C++

CHAPTER 23
EMBEDDED C++

This page was intentionally left blank

APPENDIX D INLINE FUNCTIONS

Inline member functions need to be declared so that they can be included into each module, which uses them. The simplest approach is to define the inline functions in the interface file for a class, but this has the disadvantage of exposing the implementation details of these functions to anyone reading the interface file. This appendix outlines a scheme for hiding the implementation of inline functions from the casual reader, and at the same time allowing the program to switch between non-inline and inline functions using conditional compilation. This scheme may not be applicable to all development environments.

Note that in some environments the inline function implementation file may need to be declared as "*.h" file and in other as a ".c" file. The environment may allow a neutral name as in this example.

D.1 The Interface ("*.h") file

```
//
// example.h
// interface file comment omitted
//

#ifndef example_h
#define example_h

class CExample {
public:
  // ctors and dtors
  CExample();
  CExample(const CExample & source);
  ~CExample();

  // assignment operator
  CExample & operator=(const Cexample & source);

  // function to be made inline
  int Function(/* parameters */);

protected:
  // protected member variables and unctions, if any

private:
  // private member variables and unctions, if any

};

#ifdef INLINE
#include "example.inline"
#endif // INLINE
```

```
#endif  // example_h
```

D.2 The Inline Function Implementation File

```
//
// example.inline
// implementation file comment omitted
//

#ifdef INLINE
#define inline inline
#else
#define inline
#endif // INLINE


//
// function doing something useful (hopefully)
//
inline int CExample::Function(/* parameters */) {
  // function body …
  return <int value>
}
```

D.3 The Implementation ("*.c") File

```
//
// example.c
// implementation file comment omitted
//

#include "example.h"

#ifndef INLINE
#include "example.inline"
#endif // INLINE

//
// default ctor
//

CExample::CExample() {
  // ctor body
}

// etc …
```